A static analysis for quantifying information flow in a simple imperative language*

David Clark[1], Sebastian Hunt[2], and Pasquale Malacaria[3]

[1] Department of Computer Science, Kings College, London. david@dcs.kcl.ac.uk
[2] Department of Computing, City University, London. seb@soi.city.ac.uk
[3] Department of Computer Science, Queen Mary, London. pm@dcs.qmul.ac.uk

Abstract. We propose an approach to quantify interference in a simple imperative language that includes a looping construct. In this paper we focus on a particular case of this definition of interference: leakage of information from private variables to public ones via a Trojan Horse attack. We quantify leakage in terms of Shannon's information theory and we motivate our definition by proving a result relating this definition of leakage and the classical notion of programming language interference. The major contribution of the paper is a quantitative static analysis based on this definition for such a language. The analysis uses some non-trivial information theory results like Fano's inequality and $\mathcal{L}_1$ inequalities to provide reasonable bounds for conditional statements. While-loops are handled by integrating a qualitative flow-sensitive dependency analysis into the quantitative analysis.

1 Introduction

Mathematical quantitative tools (like probability theory and statistics) have played an increasing role both in the theory and practise of most sciences. However the theory (and theory based analysis) of software systems largely relies on logics and makes little use of quantitative mathematics.

Traditionally logic is a qualitative discipline, things are true or false, provable or not, typable or not. It is our belief however that some fundamental notions in theoretical computer science might benefit from a quantitative study.

Take the notion of *interference* [9, 24] between program variables, informally the capability of variables to affect the value of other variables. Absence of interference (non-interference) is often used in proving that a system is well-behaving, whereas interference can lead to mysterious (mis-)behaviours. However the misbehaviour in the presence of interference will generally happen only when there is *enough* interference. Think in terms of electric current: non-interference between variables X, Y is the absence of a circuit involving X, Y; interference is the existence of a circuit; this however doesn't imply that there is enough "current" in the circuit to affect the behaviour of the system.

* This research partially supported by EPSRC grants EP/C545605/1, EP/C009746/1, and EP/C009967/1

Concrete examples of this are provided by *access control* based software systems. To enter such a system the user has to pass an identification stage; whatever the outcome of this stage (authorisation or failure) some information has been leaked (in the case of failure the search space for the right key has now become smaller). Hence these systems present interference [9] so they are not "secure" in a qualitative sense. However, common sense suggests to consider them secure if the interference is very small.

The aim of this paper is to use Shannon's information theory [27] to define a quantified notion of interference for a simple imperative language and to derive a program analysis based on this notion.

1.1 Attack Model

Classical covert channel analysis is motivated by military settings in which an insider has access to confidential information. The insider attempts to use some aspect of the computer system's behaviour to communicate that information to another user. In this model, the system under analysis is being viewed as a communication channel and the insider chooses inputs in such a way as to maximise the amount of information transmitted.

Our model is rather different. We consider situations in which a program has access to confidential data (the high inputs). These inputs may be controlled by the owner of the confidential data but we assume that the owner does *not* intentionally collude to leak the data. Our attacker is an outsider who may have control over the low inputs, but does not have any direct access to the confidential data or control over the high inputs. By contrast with the rather specialised military setting, our model addresses what are now far more common confidentiality problems involving every day use of the internet.

One example is the 'spyware' (trojan horse) problem: a user downloads and executes untrusted software from the internet. In this situation the user has no interest in transmitting confidential data (quite the reverse) but the software may require access to that data in order to serve the user's intended purpose. Here the user needs guarantees that software does not transmit an unacceptable amount of confidential data to any untrusted outsider.

A second example is the remote login problem: to gain access to a service, users must authenticate using a password or PIN number. In this situation the authentication process *necessarily* leaks information from the confidential password database and we need assurances that this leakage remains within acceptable limits.

1.2 Input Distributions

Our use of information theory to quantify information flow requires us to consider probability distributions on the inputs to a program. This naturally raises the question of how such distributions might be calculated and, where there is more than one "obvious" choice of distribution, how a choice might be made. Though there is no simple answer to this, we can offer a guiding principle: to give

useful results, the distribution should be a reasonable model of the attacker's uncertainty about the inputs. As a motivating example, consider the case of the login process giving access to accounts on the computer system of a small organisation. Suppose there are just four accounts (Peter, Paul, John, Ringo) and the attacker is someone trying to gain unauthorized access to one of the accounts (say John's). The high input is John's password and the low input is the attacker's guess. Passwords are 8 character sequences of ASCII characters, thus representable in 56 bits. We will assume for simplicity that the login process allows just one attempt at login. Consider two alternative scenarios:

1. The attacker has inside knowledge. He doesn't know which user has chosen which password but he does know that they have each chosen a different word from the set: black, red, orange, yellow, green, blue, indigo, violet. To this attacker, each of the eight possibilities is equally likely to be John's password, so the appropriate distribution assigns probability 1/8 to each one. This distribution has an entropy of 3 bits. Our definitions assign an information flow of just over 0.54 bits.
2. The attacker has no inside knowledge but he does know from empirical studies at other (larger) organisations, that users tend to pick certain dictionary words more often than other sequences of characters. Let us suppose (a gross simplification) that 1/8 of all users studied have been found to choose passwords uniformly from a set of 2^8 dictionary words, and the other 7/8 of users to choose uniformly from the remaining $2^{56} - 2^8$ possibilities. The entropy of this distribution is approximately $0.54 + 1 + 49 = 50.54$ bits. Our definitions assign an information flow of just over 0.006 bits.

These scenarios show that is not always best to use the 'real' distribution when analysing a system. Using the distribution from scenario 1 when attackers have only the generic knowledge of scenario 2 would lead to a needlessly pessimistic view of the system's security. In fact, both these distributions may give overly conservative results. If the company has a rigid policy of allocating automatically generated passwords, the choices may actually conform to a uniform distribution over all 2^{56} possibilities. In this case our definitions assign an information flow of less than 10^{-15} bits.

1.3 Plan of the paper

Section 2 introduces the basics of information theory and program semantics we need. We then investigate which of Shannon's measures (entropy, conditional entropy, mutual information) is the right one for the task; in 2.4 we introduce our definition of leakage and in 2.5 we establish its relation to the classical programming language definition of non-interference.

Section 3 presents a review of related work on quantifying information flow, starting from Denning's and Millen's pioneering work up to the most recent developments in the field such as Clarkson, Myers and Schneider's work.

Section 4 introduces a static analysis based on the theory presented in 2.4. Analysis rules are syntax (command and expression) based. A rule for a command provides lower and upper bounds for the leakage of one or more variables in the command. A correctness result is then proven which establishes that the bounds provided by the rules for a language statement are always consistent with the leakage of that statement according to section 2.4.

Section 5 presents improvements to the analysis of expressions and further justification for some of the analysis rules.

1.4 Relationship with previous work

In a previous paper [4] we sketched an information theory based program analysis for a simple language without loops. The present work is also related with the workshop paper [3]. In contrast to the latter work, which employs a Use Definition Graph (UDG) underlying the analysis, here we present a syntax directed analysis along the lines of that in [4]. Apart from this complete recasting of the analysis and new proofs of correctness, the significant additions in the present paper are:

theory: Formal relationships between non-interference for program variables, random variable independence and leakage of confidential data are established.

analysis of commands: The analysis of conditionals is significantly refined, using powerful results from information theory (Fano's inequality and the $\mathcal{L}_1$ inequality) to derive both upper and lower bounds on flows through if-statements.

analysis of expressions: Improved bounds on general equality tests and arithmetic expressions are presented.

In addition, the review of other significant contributions is extended and updated, all relevant proofs are provided and all sections benefit from extended discussion and more technical details and examples.

2 Information theory and interference

In this section we will provide the foundations of our work and some background on information theory.

2.1 The language and its semantics

In a semantics-based analysis of security properties, there is a trade-off between tractability and accuracy. Any semantic model is, necessarily, an abstraction of its physical implementations and the limits of an analysis based on that model are determined by the nature of the abstraction. Put more concretely, a system which can be shown to be secure with respect to a semantic model may still

be vulnerable to attacks which exploit precisely those aspects of its behaviour which are not modelled.

In this paper we consider a programming language with a simple denotational semantics and we analyse confidentiality properties based purely on the input-output behaviour which this semantics defines. The language is described in Table 1.

The guarantees provided by our analysis are correspondingly limited. In particular, our analysis addresses the question of how much an attacker may learn (about confidential information) by observing the input-output behaviour of a program, but does not tell us anything about how much can be learned from its running-time.

$$C \in \mathrm{Com} \qquad x \in \mathrm{Var} \qquad E \in \mathrm{Exp} \qquad B \in \mathrm{BExp} \qquad n \in V$$

$$C := \mathbf{skip} \mid x = E \mid C_1 \,;\, C_2 \mid \mathbf{if}\ B\ C_1\ C_2 \mid \mathbf{while}\ B\ C$$

$$E := x \mid n \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 * E_2$$

$$B := \neg B \mid B_1 \wedge B_2 \mid E_1 < E_2 \mid E_1 == E_2$$

Table 1. The Language

Given a set A, we write $A_\perp$ for the cpo obtained by the disjoint union of A and $\{\perp\}$, ordered by:

$$\perp \sqsubseteq \perp$$
$$\perp \sqsubseteq a$$
$$a \sqsubseteq a' \text{ iff } a = a'$$

where a, a' range over A (thus $a, a' \neq \perp$). If $f : A \to B_\perp$ and $g : B \to C_\perp$ we write $(g \circ f) : A \to C_\perp$ for the strict composition of g and f, thus:

$$(g \circ f)(a) = g(f(a)) \quad \text{if } f(a) \neq \perp$$
$$(g \circ f)(a) = \perp \quad \text{if } f(a) = \perp$$

The functions $A \to B_\perp$ also form a cpo when ordered pointwise.

Program variables are drawn from a finite set Var. Let V be the set of k-bit integers (that is, bit-vectors of length k interpreted as integers in the range $-2^{k-1} \leq n < 2^{k-1}$ in twos-complement representation). The set of stores Σ is just the set of functions $\sigma \in \mathrm{Var} \to V$.

An arithmetic expression E is interpreted as a function $[\![E]\!] : \Sigma \to V$, using the usual twos-complement interpretations of $+, -, *$.

A boolean expression B is interpreted as a function $[\![B]\!] : \Sigma \to \{0, 1\}$ in the standard way (0 is false, 1 is true).

A command C is interpreted as a function $[\![C]\!] : \Sigma \to \Sigma_\perp$ (see Table 2). The semantics of **while** is given as the least fix point of a function $F : (\Sigma \to \Sigma_\perp) \to (\Sigma \to \Sigma_\perp)$.

$$
\begin{aligned}
[\![\mathbf{skip}]\!] &= \lambda\sigma.\sigma \\
[\![x = E]\!] &= \lambda\sigma.\sigma[x \mapsto [\![E]\!]\sigma] \\
[\![C_1; C_2]\!] &= [\![C_2]\!] \circ [\![C_1]\!] \\
[\![\mathbf{if}\ B\ C_1\ C_2]\!] &= \lambda\sigma. \begin{cases} [\![C_1]\!]\sigma & \text{if } [\![B]\!]\sigma = 1 \\ [\![C_2]\!]\sigma & \text{if } [\![B]\!]\sigma = 0 \end{cases} \\
[\![\mathbf{while}\ B\ C]\!] &= \text{lfp } F \text{ where } F(f) = \lambda\sigma. \begin{cases} (f \circ [\![C]\!])\sigma & \text{if } [\![B]\!]\sigma = 1 \\ \sigma & \text{if } [\![B]\!]\sigma = 0 \end{cases}
\end{aligned}
$$

Table 2. Denotational Semantics

2.2 Degrees of interference

We suppose that the variables of a program are partitioned into two sets, H (*high*) and L (*low*). High variables may contain confidential information when the program is run, but these variables cannot be examined by an attacker at any point before, during or after the program's execution. Low variables do not contain confidential information before the program is run and can be freely examined by an attacker before and after (but not during) the program's execution. This raises the question of what an attacker may be able to learn about the confidential inputs by examining the low variable outputs.

One approach to confidentiality, quite extensively studied [9], is based on the notion of *non-interference*. This approach looks for conditions under which the values of the high variables have no effect on (do not 'interfere' with) the values of the low variables when the program is run. We can formalise non-interference in the current setting as follows. A terminating program P is non-interfering if, whenever $\sigma_1\ x = \sigma_2\ x$ for all x in L, then $[\![P]\!]\sigma_1 = \sigma_1'$ and $[\![P]\!]\sigma_2 = \sigma_2'$ with $\sigma_1'\ x = \sigma_2'\ x$ for all x in L. If this condition holds, an attacker learns nothing about the confidential inputs by examining the low outputs.

Thus non-interference addresses the question of *whether or not* a program leaks confidential information. In the current work, by contrast, we address the question of *how much* information may be leaked by a program.

To help explore the difference between the approaches, consider the following two programs:

1. `if (h == x) {y = 0} {y = 1}`
2. `if (h < x)  {y = 0} {y = 1}`

Here we specify that `h` is high while `x` and `y` are low. Clearly, neither of these programs has the non-interference property, since the final value of `y` is affected by the initial value of `h`. But are the programs equally effective from an attacker's point of view? Suppose we allow the attacker not only to examine but actually to *choose* the initial value of `x`. Suppose further that the attacker can run the program many times for a given choice of value for `h`. There are 2^k possible values which `h` may have and the attacker wishes to know which one it is. It is easy to see (below) that the second program is more effective than the first, but

the significance of this difference depends on the *distribution* of the values taken by `h`.

At one extreme, all 2^k values are equally likely. Using the first program it will take the attacker, on average, 2^{k-1} runs, trying successive values for `x`, to learn the value of `h`. Using the second program, the attacker can choose values of `x` to effect a binary search, learning the value of `h` in at most k runs.

At the other extreme, `h` may in fact only ever take a few of the possible values. If the attacker knows what these few values are, then both programs can clearly be used to find the actual value quickly, since the search space is small.

2.3 Background on information theory

We use Shannon's information theory to quantify the amount of information a program may leak and the way in which this depends on the distribution of inputs. Shannon's measures are based on a logarithmic measure of the unexpectedness, or surprise, inherent in a probabilistic event. An event which occurs with some non-zero probability p is regarded as having a 'surprisal value' of:

$$\log \frac{1}{p}$$

Intuitively, surprise is inversely proportional to likelihood. The base for log may be chosen freely but it is conventional to use base 2 (the rationale for using a logarithmic measure is given in [27]). The total information carried by a set of n events is then taken as the weighted sum of their surprisal values:

$$\mathcal{H} = \sum_{i=1}^{n} p_i \log \frac{1}{p_i} \tag{1}$$

(if $p_i = 0$ then $p_i \log \frac{1}{p_i}$ is defined to be 0). This quantity is variously known as the *self-information* or *entropy* of the set of events.

The events of interest for us are observations of the values of variables before and after the execution of (part of) a program. Suppose that the inputs to a program take a range of values according to some probability distribution. In this case we may use a random variable to describe the values taken (initially) by a program variable, or set of program variables.

For our purposes, a random variable is a total function $X : D \to R$, where D and R are finite sets and D comes with a probability distribution μ (D is the sample space). We adopt the following conventions for random variables:

1. if X is a random variable we let x range over the set of values which X may take; if necessary, we denote this set explicitly by $R(X)$; the domain of X is denoted $D(X)$
2. we write $p(x)$ to mean the probability that X takes the value x, that is, $p(x) \stackrel{\text{def}}{=} \sum_{d \in X^{-1}(x)} \mu(d)$; where any confusion might otherwise arise, we write this more verbosely as $P(X = x)$

3. for a vector of (possibly dependent) random variables $(X_1, \ldots, X_n)$, we write $p(x_1, \ldots, x_n)$ for the joint probability that the X_i simultaneously take the values x_i; equivalently, we may view the vector as a single random variable $<X_1, \ldots, X_n>$ with range $R(X_1) \times \cdots \times R(X_n)$
4. when summing over the range of a random variable, we write $\sum_x f(x)$ to mean $\sum_{x \in R(X)} f(x)$; again, we use the more verbose form where necessary to avoid confusion

Recall that the *kernel* of a function $f : A \to B$ is the equivalence relation $=_f$ on A defined by $a_1 =_f a_2$ iff $f(a_1) = f(a_2)$. When two random variables $X_1 : D \to R_1$ and $X_2 : D \to R_2$ have the same kernel, we say that they are *observationally equivalent*, written $X_1 \simeq X_2$.

The entropy of a random variable X is denoted $\mathcal{H}(X)$ and is defined, in accordance with (1), as:

$$\mathcal{H}(X) = \sum_x p(x) \log \frac{1}{p(x)} \tag{2}$$

It is immediate from the definitions that $X_1 \simeq X_2$ implies $\mathcal{H}(X_1) = \mathcal{H}(X_2)$ but not conversely.

Because of possible dependencies between random variables, knowledge of one may change the surprise (hence information) associated with another. This is of fundamental importance in information theory and gives rise to the notion of *conditional* entropy. Suppose that $Y = y$ has been observed. This induces a new random variable $(X|Y = y)$ (X restricted to those outcomes such that $Y = y$) with the same range as X but with domain $Y^{-1}(y)$ and $P((X|Y = y) = x) = P(X = x|Y = y)$, where

$$P(X = x|Y = y) = \frac{p(x, y)}{p(y)}$$

The conditional entropy of X given knowledge of Y is then defined as the expected value (i.e., weighted average) of the entropy of all the conditioned versions of X:

$$\mathcal{H}(X|Y) = \sum_y p(y)\mathcal{H}(X|Y = y) \tag{3}$$

A key property of conditional entropy is that $\mathcal{H}(X|Y) \leq \mathcal{H}(X)$, with equality iff X and Y are independent. Notice also the relation between conditional and joint entropy (chain rule):

$$\mathcal{H}(X, Y) = \mathcal{H}(X|Y) + \mathcal{H}(Y) \tag{4}$$

Mutual information Information theory provides a more general way of measuring the extent to which information may be shared between two sets of observations. Given two random variables X and Y, the mutual information between

X and Y, written $\mathcal{I}(X;Y)$ is defined as follows:

$$\mathcal{I}(X;Y) = \sum_x \sum_y p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \tag{5}$$

Routine manipulation of sums and logs yields three equivalent ways of defining this quantity:

$$\mathcal{I}(X;Y) = \mathcal{H}(X) + \mathcal{H}(Y) - \mathcal{H}(X,Y) \tag{6}$$
$$\mathcal{I}(X;Y) = \mathcal{H}(X) - \mathcal{H}(X|Y) \tag{7}$$
$$\mathcal{I}(X;Y) = \mathcal{H}(Y) - \mathcal{H}(Y|X) \tag{8}$$

As shown by (6), $\mathcal{I}(X;Y)$ is symmetric in X and Y.

This quantity is a direct measure of the amount of information carried by X which can be learned by observing Y (or vice versa). As with entropy, there are *conditional* versions of mutual information. The mutual information between X and Y given knowledge of Z, written $\mathcal{I}(X;Y|Z)$, may be defined in a variety of ways. In particular, (6)–(8) give rise to three equivalent definitions for $\mathcal{I}(X;Y|Z)$:

$$\mathcal{I}(X;Y|Z) = \mathcal{H}(X|Z) + \mathcal{H}(Y|Z) - \mathcal{H}(X,Y|Z) \tag{9}$$
$$\mathcal{I}(X;Y|Z) = \mathcal{H}(X|Z) - \mathcal{H}(X|Y,Z) \tag{10}$$
$$\mathcal{I}(X;Y|Z) = \mathcal{H}(Y|Z) - \mathcal{H}(Y|X,Z) \tag{11}$$

2.4 Quantifying interference using information theory

In this section we start by defining a measure of information flow appropriate in a quite general computational setting. We go on to consider the special case of flows in deterministic systems (allowing a purely functional semantics) where all inputs are accounted for, and show how the general definition simplifies in this case.

Transformational Systems A *transformational system* $\mathcal{S}$ is specified by the following:

1. $D_{\mathcal{S}}$ - a finite set (the sample space)
2. a probability distribution on $D_{\mathcal{S}}$
3. a random variable $I_{\mathcal{S}} : D_{\mathcal{S}} \to \mathcal{R}(I_{\mathcal{S}})$ which defines the inputs of the system
4. a random variable $O_{\mathcal{S}} : D_{\mathcal{S}} \to \mathcal{R}(O_{\mathcal{S}})$ which defines the outputs of the system

Note that, in any real system of interest, we would expect the outputs of the system to be determined, to some extent, by the inputs, and so $I_{\mathcal{S}}$ and $O_{\mathcal{S}}$ will not normally be independent. Note also that we could, without loss of generality, fix $D_{\mathcal{S}}$ as $\mathcal{R}(I_{\mathcal{S}}) \times \mathcal{R}(O_{\mathcal{S}})$, taking $I_{\mathcal{S}}$ and $O_{\mathcal{S}}$ to be the first and second projections, respectively. However, it is technically convenient not to do this, especially in the case of purely deterministic systems.

Given a transformational system $\mathcal{S}$, we are concerned with two classes of observation:

– An *input observation* is a surjective function $X : \mathcal{R}(I_{\mathcal{S}}) \to \mathcal{R}(X)$.
– An *output observation* is a surjective function $Y : \mathcal{R}(O_{\mathcal{S}}) \to \mathcal{R}(Y)$.

Observations induce random variables $X^{\text{in}} : D_{\mathcal{S}} \to \mathcal{R}(X)$ and $Y^{\text{out}} : D_{\mathcal{S}} \to \mathcal{R}(Y)$ by composition with the relevant component of $\mathcal{S}$:

– $X^{\text{in}} \stackrel{\text{def}}{=} X \circ I_{\mathcal{S}}$
– $Y^{\text{out}} \stackrel{\text{def}}{=} Y \circ O_{\mathcal{S}}$

Examples In the following, let N be some finite subset of the integers, such as those representable in a k-bit twos-complement representation.

1. Consider terminating programs written in a simple imperative language, defined as in Table 1 but with the addition of a `coin` operator, which evaluates randomly to 0 or 1 with equal probability. Assume given a probability distribution on input states $\sigma \in \Sigma$, writing $p(\sigma)$ for the probability that the input state is σ.
   Any such program P can be viewed as a transformational system, taking $D_{\mathcal{S}} \stackrel{\text{def}}{=} \Sigma \times \Sigma$ to be the space of input/output pairs, $I_{\mathcal{S}}(\sigma, \sigma') = \sigma$ and $O_{\mathcal{S}}(\sigma, \sigma') = \sigma'$. The probability distribution on $D_{\mathcal{S}}$ is induced by the standard semantics of P. For example, if P is the program `y := coin` then we have:
   $$\begin{array}{ll} p(\sigma, \sigma') = p(\sigma)/2 & \text{if } \sigma' = \sigma[y \mapsto 0] \text{ or } \sigma' = \sigma[y \mapsto 1] \\ p(\sigma, \sigma') = 0 & \text{otherwise} \end{array}$$
   The obvious input and output observations to consider in this setting are projections on the state, that is, observations of the values of one or more program variables. For any program variable `x`, the corresponding observation is given by $X : \sigma \mapsto \sigma(\mathtt{x})$.
2. Consider the system defined by the swap function $\lambda(x, y).(y, x)$ restricted to $N \times N$. In this case it is natural to take $D_{\mathcal{S}}$ to be $N \times N$ and $I_{\mathcal{S}}$ to be the identity. Given a probability distribution on $N \times N$, we then have a transformational system where $I_{\mathcal{S}}$ is the identity and $O_{\mathcal{S}}$ is just swap. In this case we also have $\mathcal{R}(O_{\mathcal{S}}) = N \times N = \mathcal{R}(I_{\mathcal{S}})$. Possible input and output observations include the projections π_1, π_2.

Information Flow We are interested in flows of information from system inputs to system outputs. We use some input observation $X : \mathcal{R}(I_{\mathcal{S}}) \to \mathcal{R}(X)$ and output observation $Y : \mathcal{R}(O_{\mathcal{S}}) \to \mathcal{R}(Y)$, as defined in Section 2.4, to pick out the parts of the input and output we wish to focus on. In this context, we refer to X as the *information source*. In the rest of this section, assume given some transformational system $\mathcal{S}$.

A natural information-theoretic quantity to view as the flow from X to Y is the mutual information between the two corresponding random variables: $\mathcal{I}(X^{\text{in}}; Y^{\text{out}})$. We say this seems natural because it is a direct formalisation of the idea that the quantity of information flowing from X to Y is the amount of

information given by input observation X which is shared with output observation Y.

However, despite its intuitive appeal, this formalisation is flawed as it stands. To see why, consider the case of $Y = X \text{ XOR } Z$ (true when exactly one of the arguments is true) with X and Z independent random variables uniformly distributed over the booleans. Since Y is the value of a function with argument X, and since variation in X clearly can cause variation in Y, we might expect the presence of an information flow from X to Y, But this is not shown in $\mathcal{I}(X;Y)$; indeed we have:

$$\mathcal{I}(X;Y) = \mathcal{H}(X) + \mathcal{H}(Y) - \mathcal{H}(X,Y) = 1 + 1 - 2 = 0$$

At first sight this is surprising but the explanation is straightforward: XOR is here providing perfect encryption of X, with Z as the key. An observer can learn nothing about X from Y *provided the observer does not know* Z. This shows very clearly that a satisfactory definition of flow must take account of the observer's prior knowledge of the context. The right way to do this is via *conditional* mutual information. In the XOR example, if we assume knowledge of Z and account for this by conditioning on Z, we find:

$$\mathcal{I}(X;Y|Z) = \mathcal{H}(X|Z) - \mathcal{H}(X|Y,Z)$$

Since the knowledge of the output of XOR and one of its inputs allows us to determine the other input, we can express X as a function of Y and Z, so $\mathcal{H}(X|Y,Z) = 0$. On the other hand X and Z are independent and X is a uniformly distributed Boolean so $\mathcal{H}(X|Z) = \mathcal{H}(X) = 1$ and $\mathcal{I}(X;Y|Z) = 1 - 0 = 1$.

Prior knowledge may also *decrease* the flow of information as measured by an observer. For example, an observer of the swap system who already knows Z learns nothing new about $<X,Z>$ by observing $\pi_1(\text{swap}(X,Z))$, whereas an observer who knew nothing to start with would measure a flow of $\mathcal{H}(Z)$ bits.

We therefore modify our definition of information flow accordingly. Let X and Z be input observations, let Y be an output observation. Then we define the information flow from X to Y given knowledge of Z as:

$$\mathcal{F}_Z(X \rightsquigarrow Y) \stackrel{\text{def}}{=} \mathcal{I}(X^{\text{in}}; Y^{\text{out}} | Z^{\text{in}}) \tag{12}$$

There are two natural questions arising from this basic definition:

- What is the meaning of $\mathcal{F}_Z(X \rightsquigarrow Y) = 0$? This captures the special case when *no* information flows from X to Y. In Corollary 1 we show that, in the deterministic case, this is equivalent to the standard notion of non-interference between X and Y (provided X,Z constitute the whole input).
- What is the meaning of $\mathcal{F}_Z(X \rightsquigarrow Y) = n$ for $n > 0$? By basic information theoretic inequalities we know that $n \leq \mathcal{H}(X^{\text{in}}|Z^{\text{in}})$. When n achieves this maximum, the observation is revealing everything: *all* the information in X is flowing to Y. When n falls short of the maximum, the observation is incomplete, leaving $\mathcal{H}(X^{\text{in}}|Z^{\text{in}}) - n$ bits of information still unknown. One

possible operational interpretation of this "gap" is that it provides a measure of how hard it remains to guess the actual value of X once Y is known. This is formalised in [16] where it is shown that an entropy based formula provides a lower bound for the number of guesses required by a dictionary attack (note, however, that this 'guessing game' is an idealised one in which it is assumed that the encoding of information about X in Y is invertible in constant time; it has nothing to say about the computational difficulty of recovering X from Y when this does not hold). For a more extensive discussion of this question we refer the reader to [14, 4].

Deterministic Information Flow We now restrict attention to the case of *deterministic*, transformational systems, by which we mean systems for which there exists a function f such that $O_{\mathcal{S}} = f \circ I_{\mathcal{S}}$. This will be the case, for example, in systems defined by programs written in a simple imperative language (without non-deterministic constructs) or in a functional language. Now consider flows of the form $\mathcal{F}_Z(X \rightsquigarrow Y)$ in the special case that observations X and Z jointly determine the inputs, thus $<X, Z>$ is injective. For example, in a security setting we may be interested in flows of the form $\mathcal{F}_L(H \rightsquigarrow L)$ where program variables are partitioned into the high-security set (input observation H) and the low-security set (input observation L). Such a flow measures what a low-security observer (who can only observe low-security variables) can learn about high-security inputs as a result of information flow into the low-security outputs. Since H, L partition the set of all program variables they jointly provide a complete observation of the input.

As shown by the following proposition, the deterministic case allows a simplified definition of flow:

Proposition 1. *Assume a deterministic system $\mathcal{S}$. Let X and Z be input observations and let Y be an output observation. If $<X, Z>$ is injective (thus $\mathcal{R}(I_{\mathcal{S}}) \cong P$ for some set of pairs $P \subseteq \mathcal{R}(X) \times \mathcal{R}(Z)$) then:*

$$\mathcal{F}_Z(X \rightsquigarrow Y) = \mathcal{H}(Y^{\text{out}} | Z^{\text{in}}) \tag{13}$$

Proof. By determinism and injectivity of $<X, Z>$, we have $Y^{\text{out}} = f \circ <X^{\text{in}}, Z^{\text{in}}>$, for some f. In what follows, let $A = X^{\text{in}}, B = Y^{\text{out}}, C = Z^{\text{in}}$. By the definitions (see (12) and Section 2.3), we must show $\mathcal{H}(A|C) + \mathcal{H}(B|C) - \mathcal{H}(A, B|C) = \mathcal{H}(B|C)$, that is, we must show $\mathcal{H}(A|C) = \mathcal{H}(A, B|C)$. Expanding both sides according to the definitions, we must show $\mathcal{H}(A, B, C) - \mathcal{H}(C) = \mathcal{H}(A, C) - \mathcal{H}(C)$. But $B = f \circ <A, C>$ implies $(A, B, C) \simeq (A, C)$, so we are done. □

Given its relative simplicity, it may be tempting to consider (13) as an alternative *general* definition of flow. However, in general, it is not adequate. Consider again the example program `Y := coin` from Section 2.4. Consider some other program variable `X` (the choice is arbitrary) and define $X : \sigma \mapsto \sigma(\texttt{X})$ and $Y : \sigma \mapsto \sigma(\texttt{Y})$ and let Z be any input observation. Clearly, no information flows from X to Y, since the value assigned to `Y` does not depend

on any part of the store. This is confirmed using (12), which gives a flow of $\mathcal{I}(X^{\text{in}}; Y^{\text{out}}|Z^{\text{in}}) = 0$ (one of the basic identities of information theory, since X^{in} and Y^{out} are independent). By contrast, applying (13) would give a flow of $\mathcal{H}(Y^{\text{out}}|Z^{\text{in}}) = \mathcal{H}(Y^{\text{out}}) = \frac{1}{2}\log 2 + \frac{1}{2}\log 2 = 1$.

Another striking difference between the specialised setting and the more general probabilistic setting, is that in the specialised case an upper bound on the flow into a collection of outputs can be determined by considering the outputs separately:

Proposition 2. *Let $\mathcal{S}$ be a deterministic system. Let X and Z be input observations. Let Y_1 and Y_2 be output observations and let $Y = <Y_1, Y_2>$ (thus Y is also an output observation). If $<X, Z>$ is injective then $\mathcal{F}_Z(X \rightsquigarrow Y) \leq \mathcal{F}_Z(X \rightsquigarrow Y_1) + \mathcal{F}_Z(X \rightsquigarrow Y_2)$.*

Proof. By Proposition 1, it suffices to establish the general inequality

$$\mathcal{H}(A, B|C) \leq \mathcal{H}(A|C) + \mathcal{H}(B|C) \tag{14}$$

It is easy to show (for example, by Venn diagram, see [34]) that $\mathcal{H}(A, B|C) = \mathcal{H}(A|C) + \mathcal{H}(B|C) - \mathcal{I}(A; B|C)$. Since all of the Shannon measures are non-negative, the inequality follows. □

But the conclusion of this proposition does *not* hold in general when the injectiveness condition is dropped. The reason is essentially the one used to motivate the use of conditional mutual information in the definition of flow in Section 2.4: knowledge of one input can increase the apparent flow from another. Consider the program $\lambda(x, z).(x \text{ XOR } z, z)$ defining a deterministic system with $D_{\mathcal{S}} = \mathcal{R}(I_{\mathcal{S}}) = \mathcal{R}(O_{\mathcal{S}}) = \mathit{bool} \times \mathit{bool}$. Let X and Z be the input observations π_1 and π_2, respectively. Similarly, let Y_1, Y_2 be the output observations π_1, π_2. Now suppose the distribution for $\mathcal{S}$ is such that X^{in} and Z^{in} are independent and uniform. We are concerned with flows having X as the information source. Instead of taking Z as the observer's prior knowledge (which would satisfy the injectiveness condition of the proposition) take some *constant* function W (representing ignorance), in which case, injectiveness clearly fails. Conditioning on a constant has no effect, thus $\mathcal{I}(X^{\text{in}}; Y_i^{\text{out}}|W^{\text{in}}) = \mathcal{I}(X^{\text{in}}; Y_i^{\text{out}})$, hence $\mathcal{F}_W(X \rightsquigarrow Y_i) = \mathcal{I}(X^{\text{in}}; Y_i^{\text{out}})$. Simple calculations then give the following:

- $\mathcal{F}_W(X \rightsquigarrow Y_1) = 0$
- $\mathcal{F}_W(X \rightsquigarrow Y_2) = 0$
- $\mathcal{F}_W(X \rightsquigarrow <Y_1, Y_2>) = 1$

So in this case it is *not* sufficient to calculate the flows to the outputs separately. The reason is clear: the two outputs are, respectively, a perfect encryption of X and its key. Knowing either one by itself reveals nothing about X but knowing both reveals everything.

2.5 Non-interference

In this section we consider only deterministic systems $\mathcal{S}$ and we assume the inputs and outputs to be finite functions whose domains index the components, as in the case of the store for a simple imperative language. Thus we are assuming the existence of a set of input observations $\{X_1, \ldots, X_n\}$ such that $I_{\mathcal{S}} = <X_1, \ldots, X_n>^{\text{in}}$ and a set of output observations $\{Y_1, \ldots, Y_m\}$ such that $O_{\mathcal{S}} = <Y_1, \ldots, Y_m>^{\text{out}} = f \circ <X_1, \ldots, X_n>^{\text{in}}$, for some f.

Note: it is a simple consequence of the definitions that $<X_1, \ldots, X_n>^{\text{in}} = <X_1^{\text{in}}, \ldots, X_n^{\text{in}}>$, and similarly for output observations.

In the setting of security, information flow is of particular relevance when considering confidentiality properties. Leakage of confidential information is a particular case of information flow where the source of information is a high-security part of the input and the target a low-security part of the output. In general when there is information flow from inputs to outputs, the inputs are said to *interfere* with the outputs, whereas the absence of any such flow is known as *non-interference*. One attraction of non-interference is its relative simplicity, since it is a binary property which can be defined without any explicit recourse to information theory [9]. Roughly speaking, a deterministic program is said to satisfy non-interference if its low-security outputs depend only on its low-security inputs (hence *not* on its high-security inputs).

More formally, in the deterministic case, a generalised definition of non-interference can be formalised by modelling different categories of observation (e.g. high-security, low-security) by equivalence relations. Given relations R and S, a function f is said to *map R into S*, written $f : R \Rightarrow S$, iff $\forall x, x'.(x, x') \in R \Rightarrow (f(x), f(x')) \in S$. In what follows, R and S will always be equivalence relations, though the general construction (known as logical relations [28, 22]) does not require this. Given a set of components X, let $=_X$ be the equivalence relation which relates two inputs just when they agree on all components in X. Suppose that the input (resp. output) components are partitioned into the low security components L (resp. L') and the high-security components H (resp. H'), i.e both inputs and outputs satisfy the injectivity condition with respect to H and L. Then non-interference is defined as $f : (=_L) \Rightarrow (=_{L'})$ ('low-equivalence' is mapped into 'low-equivalence'). More generally a collection of input components X *interferes* with an output component Y iff $f : (=_{\overline{X}}) \not\Rightarrow (=_Y)$, where $\overline{X}$ is the *complement* of X, i.e., all those input components *not* in X.

Here we go on to explore the relationship between non-interference and information theory.

Non-interference and Independence First recall that two random variables X and Y are independent iff for all x, y, $P(X = x, Y = y) = P(X = x)P(Y = y)$. An immediate consequence of the definition is that for two independent random variables X and Y, $\mathcal{H}(Y, X) = \mathcal{H}(Y) + \mathcal{H}(X)$, which provides a proof for the following:

Proposition 3. *Random variables X and Y are independent iff $\mathcal{I}(Y; X) = 0$.*

As the XOR example suggests, simple random variable independence is not enough to capture the absence of information flows. The correct probabilistic characterization of non-interference is via *conditional* independence:

Proposition 4. *Let Y be an output component, hence (given the assumptions of this section) $Y^{\text{out}} = f \circ <X_1, \ldots, X_n>^{\text{in}}$ for some f. Assume a probability distribution such that, for all $(x_1, \ldots, x_n)$, $P(X_1^{\text{in}} = x_1, \ldots, X_n^{\text{in}} = x_n) \neq 0$. Let $i \leq n$. Then $X_1, \ldots, X_i$ are non-interfering with Y iff*

$$\mathcal{I}(Y^{\text{out}}; X_1^{\text{in}}, \ldots, X_i^{\text{in}} | X_{i+1}^{\text{in}}, \ldots, X_n^{\text{in}}) = 0$$

Proof. The constraint that all inputs have non zero probability (i.e. $p(x_1, \ldots, x_n) \neq 0$) is to avoid f being a "constant in disguise" i.e. f could assume theoretically more than one value but in practice only one value is possible as the inputs for the other values have probability 0.

In the following we use X (resp. Z) for $X_1, \ldots, X_i$ (resp. $X_{i+1}, \ldots, X_n$).

From Proposition 1 we know that $\mathcal{I}(Y; X|Z) = \mathcal{H}(Y|Z)$ so all we have to prove is that Y, X are non-interfering iff $\mathcal{H}(Y|Z) = 0$.

($\Rightarrow$) : Y, X are non-interfering means the set of values for Y is a function of X, Z (i.e.$\mathcal{H}(Y|X, Z) = 0$) constant on the X component which implies $\mathcal{H}(Y|X, Z) = \mathcal{H}(Y|Z)$.

($\Leftarrow$) : Assume Y, X are interfering, i.e. $f(X, Z)$ is non constant on the X component. Then given only the Z component we will not know the value Y will assume, i.e. we will have uncertainty in Y given Z, i.e. $\mathcal{H}(Y|Z) > 0$. □

Corollary 1.

1. *$\mathcal{F}_{\overline{A}}(A \rightsquigarrow Y) = 0$ iff A does not interfere with Y.*
2. *When $n = 1$ we have non-interference iff $\mathcal{I}(X^{\text{in}}; Y^{\text{out}}) = 0$, that is, iff X^{in} and Y^{out} are independent random variables.*

3 A review of significant contributions

3.1 Denning's approach

In [8], Denning suggests a definition of information flow for programs, based on information theory. Given two program variables x, y in a program P and two states s, s' in the execution of P, Denning suggests that there is a flow of information from x at state s to y at state s' if uncertainty about the value of x at s given knowledge of y at s' is less than uncertainty about the value of x at s given knowledge of y at s.

Using information theory, Denning translates existence of a flow thus defined into the following condition:

$$\mathcal{H}(x_s|y_{s'}) < \mathcal{H}(x_s|y_s) \tag{15}$$

So, *if* there is a flow (i.e. (15) holds) how much information is transferred? The quantitative answer provided by Denning is a simple consequence of (15):

$$\mathcal{H}(x_s|y_s) - \mathcal{H}(x_s|y_{s'}) \tag{16}$$

i.e. the difference in uncertainty between the two situations.

A major merit of Denning's work has been to explore the use of information theory as the basis for a quantitative analysis of information flow in programs. However it doesn't suggest how the analysis could be automated. Also we argue that there is a problem with Denning's definition.

Comparison with Denning In this section we identify a flaw in Denning's definition, propose a correction, and show that the modified definition coincides with our own.

Suppose that, in state s, $y = \texttt{abs}(x)$ where x takes any integer value in the range $-16, \ldots, 15$, with uniform distribution. Then consider the following two programs:

$$(A) \quad \{s\}\texttt{if } (x = 0) \ \{y = 1\} \ \{y = 2\}\{s'\}$$

$$(B) \quad \{s\}\texttt{if } (x < 0) \ \{y = 1\} \ \{y = 2\}\{s''\}$$

Calculating the quantities involved in (16) we find:
$\mathcal{H}(x_s|y_s) \simeq 1$ (because $\mathcal{H}(x_s|y_s) \simeq \mathcal{H}(x_s|\texttt{abs}(x))$).
$\mathcal{H}(x_s|y_{s'}) \simeq 4.8$ (because $\mathcal{H}(x_s|y_{s'}) = (1/32)\mathcal{H}(x_s|y = 1). + (31/32)\mathcal{H}(x_s|y = 2) = 0 + (31/32)\texttt{log}(31)$)
$\mathcal{H}(x_s|y_{s''}) = 4$ (because $\mathcal{H}(x_s|y_{s''}) = (1/2)\mathcal{H}(x_s|y = 1) + (1/2)\mathcal{H}(x_s|y = 2) = (1/2)\texttt{log}(16) + (1/2)\texttt{log}(16)$).
Thus, according to (16), there is no flow from x to y in either case, since the uncertainty in x given y actually increases in both cases. Now it is implicit in Denning's definition that the quantity of flow depends, in part, on what is observed both before and after the program runs. The first term in (16), $\mathcal{H}(x_s|y_s)$, represents the initial uncertainty in x *given that* y *is observed*, whereas the second term is intended to represent the final uncertainty in x, again given that y is observed. The problem lies with the second term: it accounts for the final observation of y but (effectively) assumes that the initial observation has been forgotten. This is a safe assumption only if we know that the observer has no memory. In general, however, we must assume that, in the end, the observer knows *both* the initial *and* final values of y. Modifying (16) in line with this assumption, we obtain:

$$\mathcal{H}(x_s|y_s) - \mathcal{H}(x_s|y_{s'}, y_s) \tag{17}$$

Note that $\mathcal{H}(X|Y, Z) \leq \mathcal{H}(X|Y)$, for any random variables X, Y, Z, and thus (16) $\leq$ (17) will always hold.

Applying (17) to programs (A) and (B), we calculate: $\mathcal{H}(x_s|y_{s'}, y_s) = 1$ and $\mathcal{H}(x_s|y_{s''}, y_s) = 0$. So, using (17), we still find no flow for program (A) but for program (B) we have a flow of 1 bit. The crucial difference between (A) and (B) is that (A) reveals nothing *new* about x (knowing abs(x) we already know if $x = 0$) whereas (B) reveals the one thing we didn't know, namely the sign of x.

Applying (17) to the case of a program with inputs $H^{\text{in}}, L^{\text{in}}$ and outputs $H^{\text{out}}, L^{\text{out}}$, we obtain:

$$\mathcal{H}(H^{\text{in}}|L^{\text{in}}) - \mathcal{H}(H^{\text{in}}|L^{\text{out}}, L^{\text{in}}) \tag{18}$$

Proposition 5. $\mathcal{F}_L(H \rightsquigarrow L) = (18)$.

Proof. The result follows simply by rewriting both sides using the definitions of conditional entropy and conditional mutual information. □

3.2 Millen's Approach

We have seen that the work we describe in this paper is not the first attempt to apply information theory to the analysis of confidentiality properties. An early example is that of Jonathan Millen [19] which points to the relevance of Shannon's use of finite state systems in the analysis of channel capacity.

Millen was, to the best of our knowledge, the first to establish a formal correspondence between noninterference and mutual information. What he proves is that, using a state machine model, the notion of non-interference in such a system is equivalent to the mutual information between random variables representing certain inputs and outputs being equal to zero. This is hence the first result similar to Corollary 1.

Millen uses this equivalence to measure interference in state machine systems, in particular to study the channel capacity for covert channels. Millen's pioneering work is very relevant to ours; however, in contrast to Millen's work the fundamental concern of the current paper is the static analysis of programming languages.

3.3 McLean's Approach

According to McLean [18], the most stringent approach to information flow is Sutherland's Non-deducibility model [29]. This model requires High and Low objects to be effectively independent. Non-deducibility, also called compartmentalisation, may be helpful to logicians to reason about independent subsystems of a system, however it doesn't capture the notion of non-interference as intended in a security context.

McLean argues that an analysis of the notion of secure information flow, as opposed to compartmentalisation, requires the introduction of *time* into the model. When this is done, only certain classes of dependency between Low and High are considered security violations.

Figure 1 shows allowed and forbidden flows between High and Low objects at different times. The left figure expresses the fact that interference allows for $p(H_t|L_{t-1}) \neq p(H_t)$ i.e. low information can contribute to high information, whereas the right figure expresses the requirement $p(L_t|H_{t-1}) = p(L_t)$ i.e. high information cannot contribute to following low information; notice that $p(L_t|H_{t-1}) = p(L_t)$ is equivalent to $p(H_{t-1}|L_t) = p(H_{t-1})$ which justifies forbidding the upward arrow as well.

However flows in Figure 1 don't disallow the possibility that by taking the combined knowledge of L_t, L_{t-1} we may end up knowing something about H_{t-1}. To avoid this situation the requirement $p(L_t|(H_{t-1}, L_{t-1})) = p(L_t|L_{t-1})$ is introduced, illustrated by Figure 2.

The main problem with this definition of non-interference is the inability to distinguish between statistical correlation of values in the high and low objects and causal relationships between high and low objects.

McLean's model is highly abstract. System behaviours are modelled by the sequences of values taken by the 'High and Low objects' of the system. His Flow Model states that a system is secure if $p(L_t|(H_s, L_s)) = p(L_t|L_s)$, where L_t describes the values taken by the Low system objects at time t while L_s and H_s are the sequences of values taken by the Low and High objects, respectively, at times preceding t. The condition is illustrated in Figure 3.

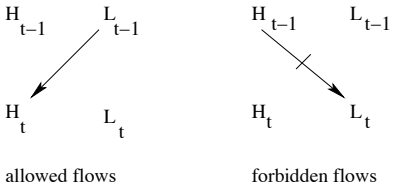

Fig. 1. allowed and disallowed flows

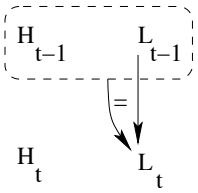

Fig. 2. safety requirement on flows

The Goguen-Meseguer approach The Goguen-Meseguer non-interference model [9] can be seen as a particular case of McLean's Flow Model, when specific assumptions about the system are made. In particular Goguen-Meseguer concentrate on deterministic programs which cannot generate High-level output from Low-level input.

McLean's Flow Model provides the right security model for a system with memory. However his work is qualitative and there is not enough machinery to implement an analysis based on it.

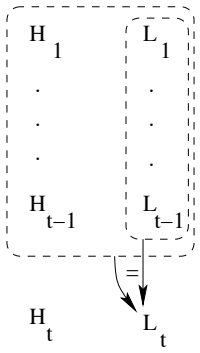

Fig. 3. McLean's Flow Model

3.4 Gray's approach

Gray models general non-deterministic systems using a similar synchronous state machine model to Millen's [19] but using probabilistic rather than non-deterministic transitions. This allows his model to include temporal and probabilistic covert channels. He assumes the system has finite sets of internal system states, communication channels, input signals and output signals as well as a single initial state. A probabilistic transition is given by a function that maps to a probability a tuple consisting of a source state, a target state, a vector of inputs indexed by the set of channels, and a vector of outputs indexed by the set of channels.

Inputs and outputs are partitioned into High and Low with the assumption that the only interaction between high and low environments is through the system via the inputs/outputs in their own partition. His model assumes that each environment has a memory of previous system inputs and outputs accessible to it.

Gray's information flow security condition is given as

$$\begin{aligned} &P_t(\alpha_L, \beta_L, \alpha_H, \beta_H) > 0 \Rightarrow \\ &\quad P_t(l_t|\alpha_L, \beta_L, \alpha_H, \beta_H) = P_t(l_t|\alpha_L, \beta_L) \end{aligned}$$

for any time t, where l_t is the Low output at time t while α_H (α_L) is the history of High (Low) inputs up to and including time $t-1$ and β_H (β_L) is the history of High (Low) outputs up to and including time $t-1$.

His system model and information flow security model can be seen as a more detailed elaboration of McLean's flow model.

Of interest from the point of view of our own work is that Gray, while mainly concerned with non-interference, is aware that his framework is sufficiently general to be used to examine non-zero leakage of classified information. Of even more interest is that he establishes a connection with information theory: he

shows that, if the flow security condition holds, the channel capacity of the channel from High to Low is zero.

He makes a definition of channel capacity for a system with memory and internal state (Shannon's original definition was for a discrete memoryless system [27]):

Definition 1 (Gray's channel capacity).
The channel capacity from H to L is

$$C \equiv \lim_{n\to\infty} C_n$$

where C_n is defined as

$$C_n \equiv \max_{\mathbf{H},\mathbf{L}} \left(\frac{1}{n} \sum_{i=1}^{n} \begin{array}{l} \mathcal{I}(\ \mathit{In\text{-}Seq\text{-}Event}_{H,i}, \\ \quad \mathit{Out\text{-}Seq\text{-}Event}_{H,i}; \\ \quad \mathit{Final\text{-}Out\text{-}Event}_{L,i} \ | \\ \quad \mathit{In\text{-}Seq\text{-}Event}_{L,i}, \\ \quad \mathit{Out\text{-}Seq\text{-}Event}_{L,i} \ ) \end{array} \right)$$

This definition says that the channel capacity is the limit of a sequence of approximations, each at a finite time n. Each approximation is calculated as the maximum over all possible High and Low behaviours of the following quantity: the average for each moment in time, i, up to the current one (n) of the mutual information between the High input and output event histories and the Low output at time i, given knowledge of the Low input and output event histories.

3.5 McIver and Morgan Approach

In their 2003 paper [17] Annabelle McIver and Carroll Morgan put forward an information theoretic security condition based on measurement of information flow for a sequential programming language enriched with probabilities. The language is the probabilistic guarded command language of [20]. The context of their discussion is program refinement and the paper establishes a number of equivalent conditions for a sequential language program to meet their security condition. There is no notion of a static analysis based on syntax.

Their security condition seeks to prevent any change in the knowledge low has of high through the operation of the program (the assumption is that the operation of the program can only increase low's knowledge of high). Low may observe low variables as well as the program text and can draw conclusions about high variables *at that point in the program*. The resulting definition of flow has a *history free* flavour. In what follows we give our version of their flow quantity definition, specialised to the case that high and low partition the store:

Definition 2 (McIver and Morgan Flow Quantity). *Let h and l be the random variables corresponding respectively to the high security and low security partitions of the store at the beginning of the program. Let h' and l' be the*

random variables corresponding to these partitions at the end of the program. The information flow from high to low is given by:

$$\mathcal{H}(h|l) - \mathcal{H}(h'|l')$$

They use this definition of flow quantity (or, as they term it, "information escape") to define a notion of channel capacity, much in the same way as Gray does, by taking the channel capacity of the program to be the least upper bound over all possible input distributions of the information flow quantity. They give both a general definition for this and a definition specialised to the case when high and low partition the store. When the channel capacity of the program is zero it is defined as secure.

They then develop some alternative formulations of this notion of security and show that these are equivalent. The alternative formulations are quite interesting. First they show that their definition means that a program can only permute high values (i.e. can't identify any of them as this will destroy entropy) as well as not communicating the value of high to low. They then show that this permutation condition is equivalent to the program preserving maximal entropy on high values (in fact a permutation of a set would preserve entropy).

Their security condition corresponding to the absence of any flows given by their definition is quite strong. In particular there does not need to be any dependency of low on high introduced by the program for a flow in their sense to exist. For example the program `h := h mod 2` will in general introduce a flow when the size of the value space of h is greater than 2, simply because the assignment will in general reduce the amount of entropy in the final value of h. Their security condition also rules out flows from low to high (which McLean calls "audit flows" and argues should be allowed) in effect achieving a separability condition, since any flow from low to high will decrease the entropy in high at program's end, given knowledge of low at program's end.

In some sense their security condition embraces both the standard definition of confidentiality (that there are no flows from a secret input to publically observable outputs) and the standard definition of integrity (that there are no flows from a possibly tainted source, low, to the untainted one, high). These have long been recognised as formal duals of each other [2]. McIver and Morgan state but do not prove a lemma that says that their security condition implies a weaker one in which the high input only is protected. It is not clear that this is correct because we feel there seem to be some problems with their definition of flow.

First, again considering a state space partitioned between high and low, if the program does not assign anything to high then their definition specialises to Denning's and suffers the same drawback: the lack of consideration of the history of low inputs. As such it can at best be a model for an attacker without memory. See the counter-example to Denning's definition above in subsection 3.1.

Second, this history free flavour can lead to some odd results. Consider swapping high and low via a third (low security) variable, temp.

```
temp := l; l := h; h := temp
```

Suppose that h and l have the same type (i.e. the same value space), are uniformly distributed, and are independent of each other. Our definition of flow quantity gives

$$\mathcal{H}(l'|l) = \mathcal{H}(l') = \mathcal{H}(h)$$

i.e. the entire secret is leaked. Since h and l are the same type and uniformly distributed their entropy is the same, i.e. $\mathcal{H}(h) = \mathcal{H}(l)$ and so their definition will give a flow of 0:

$$\mathcal{H}(h|l) - \mathcal{H}(h'|l') = \mathcal{H}(h) - \mathcal{H}(l) = 0$$

which does not mean that their results about the equivalence of the security conditions are incorrect (since the channel capacity takes the least upper bound of a non-negative set). However these two examples do call into question the usefulness of their definition of information flow.

3.6 Clarkson, Myers and Schneider approach

A recent work by Clarkson, Myers and Schneider [5] proposes a new perspective and an exciting new basis for a quantitative definition of interference. The idea is to model attacker belief about the secret input as a probability distribution. This belief is then revised using Bayesian techniques as the program is run. The attacker can be seen as a gambler and his beliefs as the amount he would be prepared to bet that the secret is such and such. This beliefs revision point of view unveils a new notion of uncertainty depending on how strongly the attacker believes something to be true.

As an illustration, the authors of [5] propose the following example: Suppose a password checking program and suppose there are three possible passwords A, B and C. The attacker believes that A is the real password (he is 99% confident about that) and he thinks B and C are equally likely with confidence 0.5% each. What happen if the attacker fails authorization using A? He is then more confused than before about what the password is, so his *uncertainty* has increased as he has now two possible passwords B and C each with a 50% chance to be the right one.

Uncertainty in terms of attacker beliefs allow for a study of the information gain in a single *experiment* (i.e. a single run of a program) and opens up investigation of flow policies for particular inputs.

One could see this approach as complementing ours. Indeed we can see our input distribution as the *bookie* view of the input. The bookie doesn't necessary know the secret (e.g. a bookie doesn't know in advance the result of a horse race) but he builds up a probabilistic model of his beliefs such that on average he can make a profit against all possible attackers. His view is statistical in nature.

3.7 Other related work

Contemporary with our own work has been that of Di Pierro, Hankin and Wiklicky. Their interest has been to measure interference in the context of a prob-

abilistic concurrent constraint setting where the interference comes via probabilistic operators. In [21] they derive a quantitative measure of the similarity between agents written in a probabilistic concurrent constraint language. This can be interpreted as a measure of how difficult a spy (agent) would find it to distinguish between the two agents using probabilistic covert channels, with a measure of 0 meaning the two agents were indistinguishable. However in contrast to our work they do not measure quantities of information. Their approach does not deal with information in an information-theoretic sense although the implicit assumption in example 4 in that paper is that the probability distribution of the value space is uniform.

Other recent works include Gavin Lowe who has measured information flow in CSP by counting refusals [13] and Volpano and Smith who have relaxed strict non-interference and developed type systems in which a well typed program will not leak its secret in polynomial time [33].

There has been also some interesting work regarding syntax directed analysis of non-interference properties. See particularly the work of Sands and Sabelfeld [25, 26].

4 Analysing programs for leakage

We now develop a system of inference rules to be used in the analysis of information flow in the simple deterministic While language defined in Section 2.1. We consider the case of security (confidentiality) properties described in Section 2.4, where the program variables are partitioned into H and L. Let $H = \{x_1, \ldots, x_n\}$ and $L = \{y_1, \ldots, y_m\}$. For vector $\mathrm{H} = <v_1, \ldots, v_n>$ we write $H = \mathrm{H}$ to mean $x_i = v_i, 1 \leq i \leq n$, and similarly for $L = \mathrm{L} = <w_1, \ldots, w_m>$.

4.1 Worst case assumptions

As described in Section 2.4 (Deterministic Information Flow), we are interested in flows of the form $\mathcal{F}_L(H \rightsquigarrow X)$ where X is the output observation for some program variable x. Since Proposition 1 applies, it suffices to calculate bounds on $\mathcal{H}(X^{\mathrm{out}}|L^{\mathrm{in}})$. However, this raises an important question about what knowledge of input distributions it is reasonable to assume. Until now we have (implicitly) assumed a probability distribution on the space of initial stores which is independent of the choice of program. There are two potential problems with this assumption:

1. while it is reasonable to assume that some knowledge will be available as to the distribution of the high inputs, it is likely that little or no knowledge will be available about the low inputs;
2. the distribution for low inputs may actually be in the *control* of the attacker; in this case it is conservative to assume that an attacker chooses L^{in} to maximise leakage.

We deal with both of these problems by constructing our analysis to give results which are safe for all possible distributions on the low inputs. The approach is, essentially, to suppose that the low inputs take some fixed (but unknown) value L. Then, rather than calculate bounds directly on $\mathcal{H}(X^{\mathrm{out}}|L^{\mathrm{in}})$, we calculate bounds on $\mathcal{H}(X^{\mathrm{out}}|L^{\mathrm{in}} = \mathrm{L})$. Our analysis calculates bounds which hold for *all* choices of L and this is conservative with respect to $\mathcal{F}_L(H \rightsquigarrow X)$, as confirmed by the following:

Proposition 6. $(\forall \mathrm{L} \,.\, a \leq \mathcal{H}(X^{\mathrm{out}}|L^{\mathrm{in}} = \mathrm{L}) \leq b) \Rightarrow a \leq \mathcal{F}_L(H \rightsquigarrow X) \leq b.$

Proof. By Proposition 1, $\mathcal{F}_L(H \rightsquigarrow X) = \mathcal{H}(X^{\mathrm{out}}|L^{\mathrm{in}})$. By definition of conditional entropy, $\mathcal{H}(X^{\mathrm{out}}|L^{\mathrm{in}})$ is the sum over all L of of $P(L^{\mathrm{in}} = \mathrm{L})\mathcal{H}(X^{\mathrm{out}}|L^{\mathrm{in}} = \mathrm{L})$. The result follows simply because a weighted average is bounded by the smallest and greatest terms. □

The analysis requires initial assumptions to be made about bounds on the values of the entropy of the input values, $\mathcal{H}(X^{\mathrm{in}}|L^{\mathrm{in}} = \mathrm{L})$. Methods by which such bounds might be calculated are beyond the scope of this paper but we can make some more or less obvious remarks:

1. Assuming k-bit variables, the bounds $0 \leq \mathcal{H}(X^{\mathrm{in}}|L^{\mathrm{in}} = \mathrm{L}) \leq k$ are always valid, though the analysis is unlikely to produce accurate results starting from such loose bounds, except in extreme cases.
2. For all low-security variables $X \in L$, $\mathcal{H}(X^{\mathrm{in}}|L^{\mathrm{in}} = \mathrm{L}) = 0$.
3. In most cases, it will be reasonable to assume that the initial values of H and L are independent, in which case $\mathcal{H}(X^{\mathrm{in}}|L^{\mathrm{in}} = \mathrm{L}) = \mathcal{H}(X^{\mathrm{in}})$ for all high-security variables $X \in H$, so the problem of calculating useful initial assumptions reduces to the problem of finding good estimates of the entropies of the high security inputs.

4.2 The analysis rules

The remainder of this section presents the syntax-directed analysis rules and gives their informal meaning and motivation. The formal statements and proofs of correctness are deferred to Section 5. We group the rules into the following five categories:

Expressions: rules dealing with boolean and arithmetic expressions.
Logical: 'logical' rules for the analysis of commands, allowing us to combine the results derived by overlapping rule sets.
Data Processing: rules expressing the basic limit placed on information flow imposed by the Data Processing theorem [6]; this category incorporates rules for a qualitative dependency analysis.
Direct Flow: rules which track simple direct flows of information due to assignment between variables and sequential control flow.
Indirect Flow: rules which deal with indirect flows arising from the influence of confidential data on the control flow through conditionals.

$$\text{EConj}\ \frac{\Gamma \vdash E : [a_1, b_1] \quad \Gamma \vdash E : [a_2, b_2]}{\Gamma \vdash E : [\max(a_1, a_2), \min(b_1, b_2)]}$$

$$\text{BConj}\ \frac{\Gamma \vdash B : [a_1, b_1] \quad \Gamma \vdash B : [a_2, b_2]}{\Gamma \vdash B : [\max(a_1, a_2), \min(b_1, b_2)]}$$

$$k\text{-Bits}\ \frac{}{\Gamma \vdash E : [0, k]} \qquad \text{1-Bit}\ \frac{}{\Gamma \vdash B : [0, 1]}$$

$$\text{Const}\ \frac{}{\Gamma \vdash n : [0, 0]} \qquad \text{Var}\ \frac{}{\Gamma, x : [a, b] \vdash x : [a, b]}$$

$$\text{And}\ \frac{\Gamma \vdash B_i : [\_, b_i] \ \ i = 1, 2}{\Gamma \vdash (B_1 \wedge B_2) : [0, b_1 + b_2]} \qquad \text{Neg}\ \frac{\Gamma \vdash B : [a, b]}{\Gamma \vdash \neg B : [a, b]}$$

$$\text{Plus}\ \frac{\Gamma \vdash E_i : [\_, b_i]}{\Gamma \vdash (E_1 + E_2) : [0, b_1 + b_2]} \qquad \text{Eq(1)}\ \frac{\Gamma \vdash E_1 : [\_, b_1] \quad \Gamma \vdash E_2 : [\_, b_2]}{\Gamma \vdash (E_1 == E_2) : [0, b_1 + b_2]}$$

$$\text{Eq(2)}\ \frac{\Gamma \vdash E_1 : [a, \_] \quad \Gamma \vdash E_2 : [\_, b]}{\Gamma \vdash (E_1 == E_2) : [0, \mathcal{B}(q)]} \quad \tfrac{1}{2^k} \leq q \leq \tfrac{1}{2},\ \mathcal{U}_k(q) \leq (a - b)$$

Table 3. Leakage inference: Expressions

4.3 Expressions

The expression analysis rules have the form $\Gamma \vdash E : [a, b]$ where Γ is a partial function from Var to real-valued pairs (representing intervals) of the form $[a, b]$ with $a \leq b$. The meaning of a rule $\Gamma \vdash E : [a, b]$ is that the expression E has leakage in the interval $[a, b]$ assuming that the leakage of each variable x in E lies in the interval $\Gamma(x)$.

The first two rules are 'logical' rules which allow us to combine derivations obtained by overlapping rule sets. For example rule [EConj] states that if we can show that the leakage of an arithmetic expression E in context Γ is $[a_1, b_1]$ and also $[a_2, b_2]$ then we can conclude that the leakage of E in the context Γ cannot be lower than $\max(a_1, a_2)$ and cannot exceed $\min(b_1, b_2)$. Rule [BConj] says the same for boolean expressions.

Rule [$k-$Bits] says that the leakage of an arithmetic expression is between 0 and k. Rule [1-Bit] says that the leakage of a boolean expression is between 0 and 1.

The leakage of a constant is 0 (rule [Const]) whereas the leakage of a variable x is given by the context $\Gamma, x : [a, b]$ (rule [Var]).

Rule [And] says that the leakage of the conjunction of two boolean expressions cannot exceed the sum of the upper bounds of the leakages of the expressions and rule [Neg] says that the leakage of the negation of a boolean expression is the same as the leakage of the original expression.

Rule [Eq(1)] is the same as rule [And]. Rule [Plus] is the same as rule [And] but applied to arithmetic expressions.

The most interesting rule is [Eq(2)]. This rule uses two functions, each of which returns an entropy given a single probability. One function is:

$$\mathcal{U}_k(q) \stackrel{\text{def}}{=} q \log \frac{1}{q} + (1-q) \log \frac{2^k - 1}{1-q} \tag{19}$$

This is easily shown to be the greatest entropy possible for any distribution on 2^k elements assuming that one element (think of this as the value of E_2) has probability q. In fact, this is the basic principle underlying Fano's inequality [6]. The other function is:

$$\mathcal{B}(q) \stackrel{\text{def}}{=} q \log \frac{1}{q} + (1-q) \log \frac{1}{1-q} \tag{20}$$

which is just the entropy of the binary distribution $\{q, 1-q\}$.

[Eq(2)] is based on the following observation: in an equality test $E_1 == E_2$, if E_1 has high entropy (lower bound a) and E_2 has low entropy (upper bound b), then the test will almost always evaluate to false. Informally, this is because the value of E_1 varies widely while the value of E_2 remains almost constant, hence their values can be equal only seldom. The probability that the test is true (q in the side condition) is therefore low. More precisely, the constraint $\mathcal{U}_k(q) \leq (a-b)$ must hold. A more detailed discussion can be found in Section 5. (We leave unstated the companion rule, justified by commutativity of ==, which reverses the roles of E_1 and E_2.)

4.4 Logical

$$\text{CConj}\frac{\vdash \Gamma\ \{C\}\ x : [a_1, b_1] \qquad \vdash \Gamma\ \{C\}\ x : [a_2, b_2]}{\vdash \Gamma\ \{C\}\ x : [\max(a_1, a_2), \min(b_1, b_2)]}$$

$$\text{Join}\frac{\vdash \Gamma\ \{C\}\ \Gamma_1 \qquad \vdash \Gamma\ \{C\}\ \Gamma_2}{\vdash \Gamma\ \{C\}\ \Gamma_1, \Gamma_2}\ \ \text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) = \emptyset$$

Table 4. Leakage inference: Logical

In general, a command leakage judgement has the form $\vdash \Gamma\ \{C\}\ \Gamma'$. When $\Gamma(x) = [a, b]$ we write $\Gamma_-(x)$ for a and $\Gamma_+(x)$ for b. The meaning of Γ is as for expressions: $\Gamma(x) = [a, b]$ asserts that, prior to execution of C, the amount of information leaked into x lies in the interval $[a, b]$. Γ' then reflects the effects of any changes to variables caused by execution of C. Note that the domains of Γ and Γ' need not be equal. In particular: many rules assert a leakage interval for just one variable on the right hand side, taking the form $\vdash \Gamma\ \{C\}\ x : [a, b]$; many rules assert either a lower bound only (intervals of the form $[a, k]$) or an upper

bound only (intervals of the form $[0, b]$). Rules [CConj] and [Join] in Table 4 then allow us to combine multiple such derivations thus constructing Γ' with larger domains and specifying smaller intervals.

Rule [CConj] is similar to [EConj], i.e. if $x : [a_1, b_1]$ and $x : [a_2, b_2]$ are derived under the same initial context and running the same command C then $x : [\max(a_1, a_2), \min(b_1, b_2)]$.

Rule [Join] states that we can join disjoint final contexts.

4.5 Data Processing

$$\text{Dep}\frac{\vdash C : \Delta}{\vdash \Gamma\ \{C\}\ x : [0, b]}\quad b = \sum_{y \in \Delta(x)} \Gamma_+(y)$$

Table 5. Leakage inference: Data Processing

The [Dep] rule relies on the following well known basic result of information theory, which simply states the more or less obvious fact that the entropy of a function of a random variable cannot exceed the entropy of the random variable itself (you can't get out more entropy from a deterministic system than you put in):

Lemma 1 (Data Processing). *Let $X_1, \ldots, X_n, Z$ be random variables with a common domain D and such that $Z = f(X_1, \ldots, X_n)$, where f is any total function. (Equivalently, in function notation: $Z = f \circ (\lambda d \in D.<X_1(d), \ldots, X_n(d)>)$.) Then $\mathcal{H}(Z) \leq \mathcal{H}(X_1, \ldots, X_n) \leq \mathcal{H}(X_1) + \cdots + \mathcal{H}(X_n)$.*

Proof. Immediate by the Data Processing Theorem [6].

The [Dep] rule is a "catch all" rule which allows us to apply this lemma wherever it may be useful. The rule makes use of a secondary derivation for qualitative dependency judgements. A dependency judgement has the form $\vdash C : \Delta$, where $\Delta : \text{Var} \to \wp(\text{Var})$. The informal meaning of a dependency judgement is as follows: if $\vdash C : \Delta$, then the value of any variable x immediately after execution of C depends on at most the values of the variables in $\Delta(x)$ immediately before execution of C. Dependency judgements are derived using the rules shown in Table 6.

In the Table we use $\Delta \sqcup \Delta'$ for the map $(\Delta \sqcup \Delta')(x) = \Delta(x) \cup \Delta'(x)$ and the condition $\Delta \sqsubseteq \Delta'$ iff for all x $\Delta(x) \subseteq \Delta'(x)$.

The general form of a derivation using these rules is $D \vdash \Delta\ \{C\}\ \Delta'$ where D is a set of program variables. Such a judgement is to be read as: in a control context which depends at most on the variables in D, if dependencies Δ hold before execution of C, then dependencies Δ' hold afterwards. A judgment $\vdash C : \Delta$ corresponds to a derivation $\emptyset \vdash \Delta_0\ \{C\}\ \Delta$ where $\Delta_0(x) = \{x\}$, for all x. In

$$\text{DSkip}\ \frac{}{D \vdash \Delta\ \{\mathbf{skip}\}\ \Delta}$$

$$\text{DAssign}\ \frac{\Delta \vdash E : D'}{D \vdash \Delta\ \{x = E\}\ \Delta[x \mapsto D \cup D']}$$

$$\text{DIf}\ \frac{\Delta \vdash B : D' \quad D \cup D' \vdash \Delta\ \{C_i\}\ \Delta'_i \quad i = 1, 2}{D \vdash \Delta\ \{\mathbf{if}\ B\ C_1\ C_2\}\ \Delta'} \quad \Delta' = \Delta'_1 \sqcup \Delta'_2$$

$$\text{DWhile}\ \frac{\Delta' \vdash B : D' \quad D \cup D' \vdash \Delta'\ \{C\}\ \Delta''}{D \vdash \Delta\ \{\mathbf{while}\ B\ C\}\ \Delta'} \quad \Delta \sqsubseteq \Delta',\ \Delta'' \sqsubseteq \Delta'$$

$$\text{DSeq}\ \frac{D \vdash \Delta\ \{C_1\}\ \Delta'' \quad D \vdash \Delta''\ \{C_2\}\ \Delta'}{D \vdash \Delta\ \{C_1\ ;\ C_2\}\ \Delta'}$$

Table 6. Dependency Analysis

some of the rules we write $\Delta \vdash E : D$ to mean that expression E has a value which depends at most on the variables in D, assuming the variable dependencies captured by Δ.

The derivation system for dependency is taken from [10], where it is shown to be a De Morgan dual of the Hoare-style independence logic of [1]. The logic of these rules is similar to that used by [7, 30, 32] but more sophisticated (and precise) in that it is flow-sensitive and thus does not require each variable to be assigned a fixed security type. The formal meaning and correctness of the rules are covered by Definition 4 and Theorem 2 below.

Notice that the only rule in our analysis which is specific to `while` statements is the [DWhile] rule in Table 6. Bounds for loops can be found by applying rule [Dep]: the bounds so obtained are conservative, i.e. it is pessimistically assumed that all that can be leaked from the variables on which the loop depends will be leaked. A more refined treatment of loops can be found in [14].

Dependency analysis: an example Consider the following Java program:

```
public static int foo(int high) {
    int n = 16;
    int low = 0;
    while (n >= 0) {
        int m = (int)Math.pow(2,n);
        if (high >= m) {
            low = low + m;
            high = high - m;
        }
        n = n - 1;
    }
    return low;
```

}

Below is a data dependency analysis for the while loop using the rules in Table 6. Notice that since in Δ' we have $\texttt{low} \mapsto \{\texttt{low}, \texttt{n}, \texttt{high}\}$ then by using rule [Dep] the analysis will conclude that all content of `high` is leaked into `low` (because $\texttt{high} \in \Delta'(\texttt{low})$). In fact, even if no direct flow is present between `high` and `low`, it can be easily seen that the above program will leak one bit of the secret at each iteration of the loop.

$$
\begin{aligned}
E &\stackrel{\text{def}}{=} 2^{\texttt{n}} \\
C &\stackrel{\text{def}}{=} C_1;\ \texttt{n} = \texttt{n} - 1 \\
C_1 &\stackrel{\text{def}}{=} \textbf{if}\ (\texttt{high} \geq E)\ C_2\ \textbf{skip} \\
C_2 &\stackrel{\text{def}}{=} \texttt{low} = \texttt{low} + E; \texttt{high} = \texttt{high} - E \\
\Delta' &\stackrel{\text{def}}{=} \Delta_0[\texttt{low} \mapsto \{\texttt{low}, \texttt{n}, \texttt{high}\}][\texttt{high} \mapsto \{\texttt{high}, \texttt{n}\}]
\end{aligned}
$$

$$
\text{DWhile}\frac{\Delta' \vdash (\texttt{n} \geq 0) : \{\texttt{n}\} \qquad \text{DSeq}\ \dfrac{\{\texttt{n}\} \vdash \Delta'\ \{C_1\}\ \Delta' \quad \{\texttt{n}\} \vdash \Delta'\ \{\texttt{n} = \texttt{n} - 1\}\ \Delta'}{\emptyset \cup \{\texttt{n}\} \vdash \Delta'\ \{C\}\ \Delta'}}{\emptyset \vdash \Delta_0\ \{\textbf{while}\ (\texttt{n} \geq 0)\ C\}\ \Delta'}
$$

where $\Delta_0 \sqsubseteq \Delta', \Delta' \sqsubseteq \Delta'$

4.6 Direct Flow

$$
\text{Assign}\frac{\Gamma \vdash E : [a, b]}{\vdash \Gamma\ \{x = E\}\ x : [a, b]} \qquad \text{Seq}\frac{\vdash \Gamma\ \{C_1\}\ \Gamma' \quad \vdash \Gamma'\ \{C_2\}\ \Gamma''}{\vdash \Gamma\ \{C_1\ ;\ C_2\}\ \Gamma''}
$$

$$
\text{Skip}\frac{}{\vdash \Gamma\ \{\textbf{skip}\}\ \Gamma} \qquad \text{NoAss}\frac{}{\vdash \Gamma, x : [a, b]\ \{C\}\ x : [a, b]}\ x \notin \text{Ass}(C)
$$

Table 7. Leakage inference: Direct Flow

Rule [Assign] states that if in a context Γ, E leaks between a and b then we deduce that after running $x = E$ under the context Γ x leaks between a and b i.e. the context $x : [a, b]$.

Rule [NoAss] states that if a variable is not assigned to inside the command C (condition $x \notin \text{Ass}(C)$) then the leakage of x after C is unchanged from Γ.

Rules [Skip] and [Seq] are standard Hoare-logic style rules for **skip** and sequential composition. We note that [Skip] is easily derived using [NoAss] and [Join] and so we do not give a direct proof of its correctness.

$$\text{If(1)}\ \frac{\Gamma \vdash B : [\_, b] \quad \vdash \Gamma\ \{C_i\}\ x : [\_, b_i]\ \ i = 1, 2}{\vdash \Gamma\ \{\mathbf{if} B\ C_1\ C_2\}\ x : [0, b + b_1 + b_2]}$$

$$\text{If(2)}\ \frac{\Gamma \vdash B : [0, 0] \quad \vdash \Gamma\ \{C_i\}\ x : [a_i, b_i]}{\vdash \Gamma\ \{\mathbf{if} B\ C_1\ C_2\}\ x : [\min(a_1, a_2), \max(b_1, b_2)]}$$

$$\text{If(3)}\ \frac{\Gamma \vdash E_1 : [a, \_] \quad \Gamma \vdash E_2 : [\_, b] \quad \vdash \Gamma\ \{C_2\}\ x : [\_, b_2]}{\vdash \Gamma\ \{\mathbf{if}(E_1 == E_2)\ C_1\ C_2\}\ x : [0, \mathcal{B}(q) + b_2 + qk]}\ \ \tfrac{1}{2^k} \leq q \leq \tfrac{1}{2},\ \mathcal{U}_k(q) \leq (a - b)$$

$$\text{If(4)}\ \frac{\vdash \Gamma\ \{C_2\}\ x : [a, \_] \quad \Gamma \vdash E_1 : [a_1, \_] \quad \Gamma \vdash E_2 : [\_, b_2]}{\vdash \Gamma\ \{\mathbf{if}(E_1 == E_2)\ C_1\ C_2\}\ x : [s(a - \mathcal{Z}(s)), k]}\ \begin{array}{l} \frac{1}{s} - s \leq \frac{1}{2}, \\ \frac{1}{2} \leq s \leq 1 - \frac{1}{2^k}, \\ \mathcal{U}_k(1 - s) = (a_1 - b_2) \end{array}$$

Table 8. Leakage inference: Indirect Flow

4.7 Indirect Flow

Rule [If(1)] states that the leakage of x after a conditional cannot exceed the sum of the leakage of the guard and the leakage of x in both branches (a direct consequence of the Data Processing Lemma).

It may seem at first sight that this rule is too conservative and perhaps should be replaced by a rule that calculates an upper bound of $b + \max(b_1, b_2)$. The following example demonstrates why $b + \max(b_1, b_2)$ is not an upper bound in general.

```
if (h < 2) {
    if (h == 0) x = 0; else x = 1;
} else {
    if (h == 2) x = 2; else x = 3;
}
```

If `h` is uniformly distributed over $\{0, 1, 2, 3\}$ this program leaks all 2 bits of information in `h` to `x`.

Our analysis would use a conservative upper bound of 1 bit for the leakage due to `h < 2` and we can precisely calculate the leakage due to the two tests `h == 0` and `h == 2` as $1/4 \log 4 + 3/4 \log(4/3) = 0.8113$ for each. So an upper bound on the leakage using $b + \max(b_1, b_2)$ produces 1.8113 which is not safe. The upper bound calculated by [If(1)] is $0.8113 + 0.8113 + 1 = 2.623$.

Rule [If(2)] states that if there is no indirect flow in a conditional then the leakage of x after the conditional is the worst possible choice of leakage along the two branches.

Rule [If(3)] states that if a conditional has as guard an equality test then the upper bound of the leakage of x after the command is given by summing:

1. The upper bound for the equality test as given by [Eq(2)].
2. The upper bound of the leakage of x in the *false*-branch.

3. The maximum leakage possible (k) multiplied by the upper bound for the probability of the test being true (q).

The idea in using qk for the true-branch is that if the test is true very seldom then q will be very small and so qk will be a reasonable approximation to the leakage of the true-branch.

Rule [If(4)] aims to provide lower bounds for conditionals when the guard is an equality test. It says that if s is a lower bound of the probability of the test being false and a is lower bound for the leakage of x in the *false*-branch, then a lower bound for the leakage of x after the conditional is given by $s(a - \mathcal{Z}(s))$ where $\mathcal{Z}(s) = (\frac{1}{s} - s)(k\ \log(\frac{1}{s} - s))$ will be formally justified later. The quantity $a - \mathcal{Z}(s)$ is a lower bound on the entropy of x in C_2 given knowledge that the equality test is false. As shown later this lower bound is a consequence of the information theoretic L_1 inequality.

5 Correctness

Correctness of the analysis is stated with respect to a non-standard "angelic" denotational semantics in which non-termination is replaced by the behaviour of **skip**. In Section 5.3 we show that, for the purposes of calculating leakage, the angelic semantics gives a tight approximation to the standard semantics, in the sense that it underestimates the overall leakage by at most 1 bit.

The angelic semantics of a command C is denoted $[\![C]\!]^A : \Sigma \to \Sigma$ and is defined, like the standard semantics, by induction on the syntax. The definition is formally identical to the standard semantics with the exception of the case for **while**, which is as follows:

$$
\begin{array}{l}
[\![\textbf{while}\ B\ C]\!]^A = f^A \text{ where:} \\
\quad f^A(\sigma) = \begin{cases} f(\sigma) & \text{if } f(\sigma) \neq \bot \\ \sigma & \text{if } f(\sigma) = \bot \end{cases} \\
\quad f = \text{lfp}\ F \\
\quad F(f) = \lambda\sigma. \begin{cases} f([\![C]\!]^A\sigma) & \text{if } [\![B]\!]\sigma = 1 \\ \sigma & \text{if } [\![B]\!]\sigma = 0 \end{cases}
\end{array}
$$

The angelic semantics 'over approximates' the standard semantics, in the following sense:

Lemma 2. *For all C, σ, if $[\![C]\!]\sigma \neq \bot$ then $[\![C]\!]\sigma = [\![C]\!]^A\sigma$.*

Proof. By a simple induction on C, observing that $[\![C]\!]^A$ can trivially be viewed as a map of type $\Sigma \to \Sigma_\bot$, so the lemma says that $[\![C]\!]^A \sqsupseteq [\![C]\!]$, then appeal to monotonicity of lfp. □

Correctness then amounts to the following: if $\vdash \Gamma\ \{C\}\ \Gamma'$ then, for all input distributions which model Γ, the output distribution (as determined by the angelic semantics) models Γ'. To make this precise, let $G : \Sigma \to V$ (think of G as the semantics of some expression) and let $F : \Sigma \to \Sigma$ (think of F as the

angelic semantics of some initial part of the program being analysed). In the following we assume given a distribution on Σ, so that any such G and F may be considered as random variables.

Definition 3. *Say that G models $[a,b]$, written $G \models [a,b]$, if $a \leq \mathcal{H}(G|L^{\text{in}} = \text{L}) \leq b$ for all* L. *Say that F models Γ, written $F \models \Gamma$, if $\lambda\sigma.F(\sigma)(x) \models \Gamma(x)$ for all x in the domain of Γ. We write $\models \Gamma$ as shorthand for $\lambda\sigma.\sigma \models \Gamma$.*

Informally $\lambda\sigma.F(\sigma)(x)$ maps a store σ into the value of the variable x in the store $F(\sigma)$. We will use $\hat{F}_x$ (or simply $\hat{F}$ when x is clear from the context)for $\lambda\sigma.F(\sigma)(x)$. Similarly for $G : \Sigma \to V$, $\hat{G}$ will stand for $\lambda\sigma.G(\sigma)$. To simplify notations we will often write $\hat{C}$ for a command C instead of $\widehat{[\![C]\!]^A}$. For example $\widehat{z = 3}_x$ is the map which send a store σ into $\sigma(x)$ if $x \neq z$ and into 3 if $x = z$.

We can now state correctness of the analysis rules.

Theorem 1 (Correctness).

1. *If $F \models \Gamma$ and $\Gamma \vdash E : [a,b]$ then $[\![E]\!] \circ F \models [a,b]$.*
2. *If $F \models \Gamma$ and $\vdash \Gamma\ \{C\}\ \Gamma'$ then $[\![C]\!]^A \circ F \models \Gamma'$.*

Corollary 2. *If $\models \Gamma$ and $\vdash \Gamma\ \{C\}\ \Gamma'$ then, for the transformational system induced by $[\![C]\!]^A$, $\Gamma'(x) = [a,b] \Rightarrow a \leq \mathcal{F}_L(H \rightsquigarrow X) \leq b$, where $X = \lambda\sigma.\sigma(x)$.*

In the proofs below we suppress explicit mention of the assumptions $L^{\text{in}} = \text{L}$. The justification for this is that $\mathcal{H}(X|L^{\text{in}} = \text{L})$ is just $\mathcal{H}(X)$ for the residual distribution given $L^{\text{in}} = \text{L}$ and we are implicitly quantifying over all possible choices of L. We write, for example, $\mathcal{H}(X) \geq b$, as shorthand for $\forall\text{L}.\mathcal{H}(X|L^{\text{in}} = \text{L}) \geq b$, and so on.

The proof of the Correctness Theorem is by induction on the height of the derivations. The proof cases for Part 1 are very simple, except for the rule [Eq(2)] which is dealt with in Section 5.1.

Proof (Correctness, Part 1). Note throughout that 0 is necessarily a correct lower bound since entropy is non-negative.

Case: EConj, BConj. Immediate by inductive hypothesis.

Case: k-Bits, 1-Bit. These are absolute upper bounds determined by the sizes of the value spaces for expressions in the language.

Case: Const. The entropy of any constant function is 0.

Case: Var. Immediate from the definitions.

Case: And, Plus, Eq(1). Immediate by the Data Processing Lemma.

Case: Neg. Permutation of the elements in a domain has no effect on entropy.

Case: Eq(2). See Section 5.1.

□

Before proving part 2 of the Correctness Theorem, we need to establish the correctness of the Dependency Analysis (Table 6). The property we require is that $\vdash C : \Delta$ implies $\lambda\sigma.[\![C]\!]^A\sigma(x)$ can be decomposed as $G \circ F$ where F picks out just the vector of variables in $\Delta(x)$. We derive this as a corollary of a more general property of the Dependency Analysis, formalised using the following definition (which may be viewed as a qualitative analogue of Definition 3).

Definition 4. *Let $D \in \wp(\mathrm{Var})$. Say that $G : \Sigma \to V$ is determined by D if $(\forall x \in D.\sigma(x) = \sigma'(x)) \Rightarrow G(\sigma) = G(\sigma')$. Say that $F : \Sigma \to \Sigma$ is determined by Δ if $\lambda\sigma.F(\sigma)(x)$ is determined by $\Delta(x)$ for all x.*

Theorem 2. *Suppose $D \vdash \Delta\ \{C\}\ \Delta'$. Then:*

1. *For all program variables x, if $\exists\sigma,\sigma'.[\![C]\!]^A\sigma x \neq [\![C]\!]^A\sigma' x$, then $\Delta'(x) \supseteq D$.*
2. *For all $F : \Sigma \to \Sigma$, if F is determined by Δ then $[\![C]\!]^A \circ F$ is determined by Δ'.*

Corollary 3. *Suppose $\vdash C : \Delta$ and $\Delta(x) = \{x_1, \ldots, x_n\}$. Then there exists $G : V^n \to V$ such that $\lambda\sigma.[\![C]\!]^A\sigma(x) = G \circ (\lambda\sigma.<\sigma(x_1), \ldots, \sigma(x_n)>)$.*

Proof. Proof of the theorem is by induction on C. This is essentially the same as the correctness proof in [1]. □

Proof (Correctness, Part 2).

Case: Dep. We must show $\mathcal{H}(\lambda\sigma.([\![C]\!]^A \circ F)\sigma(x)) \leq b$, where $b = \sum_{y\in\Delta(x)} \Gamma_+(y)$. First note that $\lambda\sigma.([\![C]\!]^A \circ F)\sigma(x) = (\lambda\sigma.[\![C]\!]^A\sigma(x)) \circ F$. and, by Corollary 3, $\lambda\sigma.[\![C]\!]^A\sigma(x) = G \circ (\lambda\sigma.<\sigma(y_1), \ldots, \sigma(y_n)>)$, where $\{y_1, \ldots, y_n\} = \Delta(x)$. Thus:

$$\begin{aligned}
&\lambda\sigma.([\![C]\!]^A \circ F)\sigma(x)\\
&= G \circ (\lambda\sigma.<\sigma(y_1), \ldots, \sigma(y_n)>) \circ F\\
&= G \circ (\lambda\sigma.<F\sigma(y_1), \ldots, F\sigma(y_n)>)\\
&= G \circ (\lambda\sigma.<(\lambda\tau.F\tau(y_1))(\sigma), \ldots, (\lambda\tau.F\tau(y_n))(\sigma)>
\end{aligned}$$

Thus, by the Data Processing Lemma, $\mathcal{H}(\lambda\sigma.([\![C]\!]^A \circ F)\sigma(x)) \leq \mathcal{H}(\lambda\tau.F\tau(y_1)) + \cdots + \mathcal{H}(\lambda\tau.F\tau(y_n))$ By assumption, $F \models \Gamma$, hence $\mathcal{H}(\lambda\tau.F\tau(y_i)) \leq \Gamma_+(y_i)$, for $1 \leq i \leq n$, as required.

Case: CConj. The hypothesis is $F \models \Gamma$ implies $H \models x : [a_i, b_i]$ $(i = 1, 2)$ where $H = [\![C]\!]^A \circ F$. That means $a_i \leq \mathcal{H}(\hat{H}) \leq b_i$, $i = 1, 2$, hence $\mathtt{min}(a_1, a_2) \leq \mathcal{H}(\hat{H}) \leq \mathtt{max}(b_1, b_2)$.

Case: Join. The hypothesis is $F \models \Gamma$ implies $H \models \Gamma_i$, for $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$ and $i = 1, 2$ where $H = [\![C]\!]^A \circ F$. This means that for all $x \in \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2)$ $\Gamma_-(x) \leq \mathcal{H}(\hat{H}) \leq \Gamma_+(x)$ and so $H \models \Gamma_1, \Gamma_2$.

Case: Assign. By induction hypothesis $F \models \Gamma$ implies $H \models [a, b]$ where $H = [\![E]\!] \circ F$. But by definition of semantics of assignment $\hat{H} = \lambda\sigma.[\![x = E]\!]^A \circ F(\sigma)(x)$ and so $[\![x = E]\!]^A \circ F \models x : [a, b]$.

Case: Seq. By induction hypothesis $F \models \Gamma$ implies $[\![C_1]\!]^A \circ F \models \Gamma'$ and $F' \models \Gamma'$ implies $[\![C_2]\!]^A \circ F' \models \Gamma$", hence $F \models \Gamma$ implies $[\![C_2]\!]^A \circ [\![C_1]\!]^A \circ F \models \Gamma''$ that is $[\![C_1; C_2]\!]^A \circ F \models \Gamma''$.

Case: NoAss. Assuming $F \models \Gamma, x : [a, b]$ and C doesn't change the value of x we want $[\![C]\!]^A \circ F \models x : [a, b]$. Notice that for all stores σ, $[\![C]\!]^A \circ F(\sigma)(x) = F(\sigma)(x)$ (because $x \notin \texttt{Ass}(C)$) and so we are done by using the induction hypothesis $F \models \Gamma, x : [a, b]$.

Case: If(1). Here the hypothesis is $F \models \Gamma$ implies $[\![B]\!] \circ F \models [-, b]$ and $[\![C_i]\!] \circ F \models [-, b_i]$ for $i = 1, 2$. The definition of the `if` statement in the denotational semantics is as a function `IF` of the semantics of the guard and the branches, i.e. $[\![\texttt{if } B\ C_1\ C_2]\!]^A = \texttt{IF}([\![B]\!], [\![C_1]\!]^A, [\![C_2]\!]^A)$ and by compositionality of the semantics $[\![\texttt{if } B\ C_1\ C_2]\!]^A \circ F = \texttt{IF}([\![B]\!] \circ F, [\![C_1]\!]^A \circ F, [\![C_2]\!]^A \circ F)$. We hence apply the Data Processing theorem to conclude:

$$\begin{aligned} \mathcal{H}(\lambda\sigma.\ [\![\texttt{if } B\ C_1\ C_2]\!]^A \circ F(\sigma)(x)) = \\ \mathcal{H}(\lambda\sigma.\texttt{IF}([\![B]\!] \circ F, [\![C_1]\!]^A \circ F, [\![C_2]\!]^A \circ F)(\sigma)(x)) \leq \\ \mathcal{H}(\lambda\sigma.[\![B]\!] \circ F(\sigma)) + \mathcal{H}(\lambda\sigma[\![C_1]\!] \circ F(\sigma)(x)) + \mathcal{H}(\lambda\sigma.[\![C_2]\!] \circ F(\sigma)(x)) \leq \\ b + b_1 + b_2 \end{aligned}$$

Case: If(2). By applying the induction hypothesis to the premise $\Gamma \vdash B : [0, 0]$ i.e. $F \models \Gamma$ implies $\mathcal{H}(\lambda\sigma.[\![B]\!] \circ F(\sigma)) = 0$ we deduce (by basic information theory) that, for each choice of a low input, the map $[\![B]\!]$ is a constant function. Hence for each choice of a low input $\texttt{IF}([\![B]\!], [\![C_1]\!]^A, [\![C_2]\!]^A)$ is either $[\![C_1]\!]^A$ or $[\![C_2]\!]^A$ and hence the result follows.

Case: If(3), If(4). Correctness for these rules is proved in section 5.2. □

5.1 Analysis of equality tests

This section will justify rule [Eq(2)] i.e. analysis of tests of the form $E_1 == E_2$. By associating random variables X and Y to the two expressions, this becomes a problem of refining bounds on $\mathcal{H}(X = Y)$ (by $\mathcal{H}(X = Y)$ we mean $\mathcal{H}(Z)$ where $Z =$ true if $X = Y$, $Z =$ false otherwise).

We begin with a simple observation: when the distribution of values for E_1 is close to uniform (X has high entropy) and the distribution for E_2 is concentrated on just a few values (Y has low entropy), then *most of the time*, E_1 and E_2 will not be equal. Thus, qualtitatively, we can see that high entropy for X and low entropy for Y implies a low probability that $E_1 == E_2$ evaluates to true (a low value for $P(X = Y)$). If we can quantify this bound on $P(X = Y)$ we can use it to calculate a bound on $\mathcal{H}(X = Y)$.

It turns out that a well known result from information theory provides just such a quantitative bound:

Lemma 3 (Fano's inequality, [6]). *Let X and Y be random variables taking values in a set of size 2^k. Then*

$$\mathcal{H}(X|Y) \leq \mathcal{U}_k(P(X = Y))$$

where $\mathcal{U}_k(q) \stackrel{\text{def}}{=} q \log \frac{1}{q} + (1-q) \log \frac{2^k - 1}{1-q}$.

We note that Fano's inequality is normally stated in terms of the so called *error probability* P_e, where $P_e \stackrel{\text{def}}{=} P(X \neq Y)$. Since $P_e = 1 - P(X = Y)$, our presentation is a trivial rewriting of the usual one.

Our justification of [Eq(2)] proceeds in three steps:

1. [Lemma 4] We argue that the quantity $a - b$ used in the side condition of [Eq(2)] is a lower bound for $H(X|Y)$, hence $\mathcal{U}_k(q) \leq a - b$ implies $\mathcal{U}_k(q) \leq H(X|Y)$.
2. [Lemma 5] We use Fano's inequality to show that any $q \geq 1/2^k$ such that $\mathcal{U}_k(q) \leq H(X|Y)$ is an upper bound for $P(X = Y)$.
3. [Lemma 6] For $q \leq 0.5$, if q is an upper bound for $P(X = Y)$ then $\mathcal{B}(q)$ is an upper bound for $\mathcal{H}(X = Y)$.

Lemma 4. *Let $a \leq \mathcal{H}(X)$ and let $b \geq \mathcal{H}(Y)$. Then $a - b \leq \mathcal{H}(X|Y)$.*

Proof. By assumption $a \leq \mathcal{H}(X)$ and $b \geq \mathcal{H}(Y)$, hence $a - b \leq \mathcal{H}(X) - \mathcal{H}(Y)$. Thus, since $\mathcal{H}(X) \leq \mathcal{H}(X, Y)$, $a - b \leq \mathcal{H}(X, Y) - \mathcal{H}(Y) = \mathcal{H}(X|Y)$. □

Lemma 5. *Let X and Y be random variables taking values in a set of size 2^k and let $q \geq 1/2^k$. Then $\mathcal{U}_k(q) \leq H(X|Y)$ implies $q \geq P(X = Y)$.*

Proof. We show the contrapositive. Suppose that $P(X = Y) > q$. Note that for $q \geq 1/2^k$, $\mathcal{U}_k(q)$ is a decreasing function of q (see, for example, figure 4) hence $P(X = Y) > q$ implies $\mathcal{U}_k(P(X = Y)) < \mathcal{U}_k(q)$. By Fano's inequality, $H(X|Y) \leq \mathcal{U}_k(P(X = Y))$, hence $H(X|Y) < \mathcal{U}_k(q)$, as required. □

Lemma 6. *Let $q \leq 0.5$. Then $q \geq P(X = Y)$ implies $\mathcal{B}(q) \geq \mathcal{H}(X = Y)$.*

Proof. $\mathcal{B}(q)$ is an increasing function of q in the region $0 \leq q \leq 0.5$ (see figure 4). □

Together, these three results provide:

Proposition 7 (Correctness of [Eq(2)]). *If $a \leq \mathcal{H}(X)$ and $\mathcal{H}(Y) \leq b$ then $\mathcal{H}(X = Y) \leq \mathcal{B}(q)$ for any $1/2^k \leq q \leq 0.5$ such that $\mathcal{U}_k(q) \leq a - b$.*

When using [Eq(2)], as the third lemma above shows, smaller values for q give tighter upper bounds for $\mathcal{H}(X = Y)$. So to find the best upper bound we need to solve equations of the form $\mathcal{U}_k(q) - (a - b) = 0$, where a and b are known values. For this, simple numerical techniques suffice [12]. Another computationally useful fact is that $\mathcal{B}(q) + (1-q)k$ is an upper bound for $\mathcal{U}_k(q)$ and that this bound is very tight unless k is small. We note that [Eq(2)] will give useful results in the case that a is high and b is low, that is, when E_1 is

known to contain a large amount of confidential information and E_2 is known to contain very little.

The way in which rule [Eq(2)] can be applied is illustrated by the example shown in fig. 4. This plots $\mathcal{U}_k(q)$ and $\mathcal{B}(q)$ against q for $k = 4$ and shows that for a lower bound of $(a - b) = 3.75$, q is bounded by $0 \leq q \leq 0.25$ (the precise upper bound is slightly lower than this).

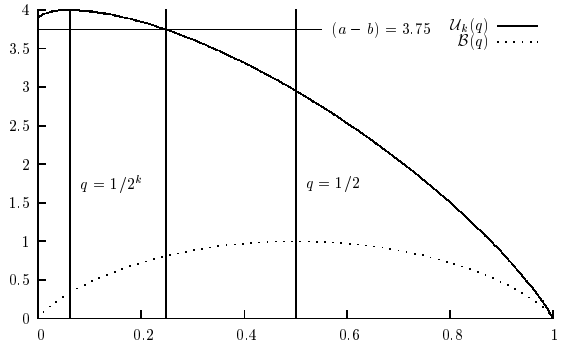

Fig. 4. the upper entropy for q in 4 bits

Example1: Consider the program P:

```
if (h == 0) x = 0; else x = 1;
```

with `h` high-security.

Suppose that $k = 32$ and the input distribution makes `h` uniform over its 2^{32} possible values; by definition of entropy of the uniform distribution we can therefore analyse the program starting with the assumption $\mathcal{H}(\mathtt{h}) = \log\ 2^{32} = 32$. We can then derive that the leakage from `h` to `x` will be in the interval $[0, 0.78 \times 10^{-9}]$. Let Γ be such that $\Gamma(h) = [32, 32]$ and let $\epsilon \stackrel{\text{def}}{=} 0.78 \times 10^{-9}$. The derivation is illustrated in Table 9. The use of [Eq(2)] is justified by checking the side condition for $q = \frac{1}{2^{32}}, a = 32, b = 0$ (check $\mathcal{U}_k(q) = 32 = 32 - 0$) and $\mathcal{B}(q) = \epsilon$. Derivation Δ_0 is the sub-derivation for the true-branch $x = 0$:

$$\text{Assign}\frac{\text{Const}\ \dfrac{}{\Gamma \vdash 0 : [0,0]}}{\vdash \Gamma\ \{x = 0\}\ x : [0,0]}$$

Derivation Δ_1, the sub-derivation for the false-branch $x = 1$, is similar.

Example 2: Consider the following schema for Java programs using n if statements, where each `Bi(high)` is a boolean expression which tests the ith bit of

$$\text{If(1)}\ \dfrac{\text{Eq(2)}\ \dfrac{\text{Var}\ \dfrac{}{\Gamma \vdash h : [32,32]} \quad \text{Const}\ \dfrac{}{\Gamma \vdash 0 : [0,0]}}{\Gamma \vdash (h == 0) : [0,\epsilon]} \quad \Delta_0 \quad \Delta_1}{\vdash \Gamma\ \{\mathbf{if}(h == 0)\ (x = 0)\ (x = 1)\}\ x : [0, \epsilon + 0 + 0]}$$

Table 9. Derivation for Example 1

`high`. Such programs thus copy n bits of `high` (a 32 bit variable) into *low* and then test `high` and `low` for equality. We assume $n < 32$. Note that `high` itself is not modified.

What will be the result of the analysis on the final test `low` == `high`? Assuming that the `high` values are uniformly distributed the result depends on n. By using the techniques described in the previous pages we can bound the leakage; a few values are shown in Table 10. We have assumed uniform distribution, hence $\mathcal{H}(\texttt{high}) = 32$ and at the end of the program $\mathcal{H}(\texttt{low}) = n$. In order to upper bound $\mathcal{H}(\texttt{low} == \texttt{high})$ we use [Eq(2)], i.e. we first determine q such that $\mathcal{H}(\texttt{high}) - \mathcal{H}(\texttt{low}) = \mathcal{U}_{32}(q)$ and we then compute $\mathcal{B}(q)$.

```
public static void foo(int high)
{
    int low = 0;
    int b;
    if (B1(high)) b = 2^0; else b = 0;
    low = low + b;
    ...
    if (Bn(high)) b = 2^(n-1); else b = 0;
    low = low + b;
    System.out.println(low == high);
}
```

Table 10.

n	$\mathcal{H}(\texttt{high}) - \mathcal{H}(\texttt{low})$	q	$\mathcal{B}(q) \geq \mathcal{H}(\texttt{low} == \texttt{high})$
2	30	0.074	0.38
7	25	0.24	0.80
12	20	0.41	0.98

5.2 Bounding leakage into a variable via execution of an if statement

This section provides justifications for rules [If(3)] and [If(4)].

We can set reasonable bounds on the leakage into a given variable as the result of the execution of an if statement (implicitly, this is a variable within the statement), assuming that we have the following:

- bounds on the leakage due to evaluation of the control expression and
- bounds on the leakage into the variable as a result of the execution of the individual branches.
- the control expression is an equality test
- the test is not true very often (i.e. the probability $p(\hat{e} == \texttt{ff}) \approx 1$)

Notation-wise in the following pages we are referring to the following command:
$\texttt{C} = \texttt{if e } \texttt{C}_1 \texttt{ C}_2$

Upper Bounds From standard information theory

$$\mathcal{H}(\hat{C}) \leq \mathcal{H}(\hat{C}, \hat{e}) = \mathcal{H}(\hat{C}|\hat{e}) + \mathcal{H}(\hat{e})$$

Assuming we can calculate an upper bound for $\mathcal{H}(\hat{e})$ as shown in the previous section, we want to find a bound on the other term, $\mathcal{H}(\hat{C}|\hat{e})$. Since we will be using the quantity $p(\hat{e} == \texttt{ff})$ extensively in what follows we give it a name, r. That is, $p(\hat{e} == \texttt{ff}) \stackrel{\text{def}}{=} r$.

We can expand $\mathcal{H}(\hat{C}|\hat{e})$ into a term summing the weighted contribution from each branch:

$$\begin{aligned}\mathcal{H}(\hat{C}|\hat{e}) &= r\mathcal{H}(\hat{C}|\hat{e} == \texttt{ff}) + (1-r)\mathcal{H}(\hat{C}|\hat{e} == \texttt{tt}) \\ &= r\mathcal{H}(\hat{C}_2|\hat{e} == \texttt{ff}) + (1-r)\mathcal{H}(\hat{C}_1|\hat{e} == \texttt{tt})\end{aligned}$$

We assume the entropy of the true-branch is bounded by k (the maximum entropy possible for x, assuming x is a k-bit variable, and the always-available worst case assumption). Since r is close to 1 this branch will make a small contribution when weighted. We can show the weighted other branch is bounded by $\mathcal{H}(\hat{C}_2)$.

$$\begin{aligned}r\mathcal{H}(\hat{C}_2|\hat{e} == \texttt{ff}) &\leq r\mathcal{H}(\hat{C}_2|\hat{e} == \texttt{ff}) + (1-r)\mathcal{H}(\hat{C}_2|\hat{e} == \texttt{tt}) \\ &= \mathcal{H}(\hat{C}_2|\hat{e}) \\ &\leq \mathcal{H}(\hat{C}_2)\end{aligned}$$

Hence we have

$$\mathcal{H}(\hat{C}) \leq \mathcal{H}(\hat{e}) + \mathcal{H}(\hat{C}_2) + (1-r)k$$

The inference rule for this is given as rule [If(3)] in Table 8.

Lower Bounds We discuss setting a lower bound on the leakage into the variable. Given the role that lower bounds play in determining upper bounds in the analysis of equality tests (see above), it is quite useful to be able to determine tight lower bounds. To bound the leakage from below we use the $\mathcal{L}_1$ inequality proposition (this is well known in information theory and can be found in [6]):

We state the proposition. First, we need to define the $\mathcal{L}_1$ distance between two probability distributions on the same event space.

Definition 5 ($\mathcal{L}_1$ Distance). *Define the $\mathcal{L}_1$ distance between two probability distributions, p and q on the same event space, written $\|p-q\|_1$, as follows*

$$\|p-q\|_1 \stackrel{\text{def}}{=} \sum_v |p(v)-q(v)|$$

The following is proved in [6] (Theorem 16.3.2)

Proposition 8 ($\mathcal{L}_1$ Inequality).

$$\|p-q\|_1 \leq \frac{1}{2} \Rightarrow |\mathcal{H}(p)-\mathcal{H}(q)| \leq \|p-q\|_1 \log\left(\frac{|X|}{\|p-q\|_1}\right)$$

where $|X|$ is the size of the event space for both p and q.

We can manipulate the RHS of the inequality in the consequent as follows:

$$\begin{aligned}\|p-q\|_1 \log\left(\frac{|X|}{\|p-q\|_1}\right) &= \|p-q\|_1(\log|X| - \log\|p-q\|_1) \\ &= \|p-q\|_1(k - \log\|p-q\|_1)\end{aligned}$$

Let's now define the two following probability distributions:

- $p(v) = P(\hat{C}_2 = v)$ i.e. $\Sigma_{\sigma:\hat{C}_2\sigma=v} p(\sigma)$
- $q(v) = P(\hat{C}_2 = v | \hat{e} == \texttt{ff})$ i.e. $\Sigma_{\sigma:\hat{C}_2\sigma=v} p(\sigma | \hat{e} == \texttt{ff})$

Hence $p(v)$ is the sum of the probabilities of all stores where after the evaluation of C_2, $x = v$ and $q(v)$ is the sum of the probabilities of all stores where after the evaluation of C_2, $x = v$ conditioned to such stores having e evaluated to false.

Recall that we assume $p(\hat{e} == \texttt{ff}) \approx 1$.

We now show that $\|p-q\|_1 \leq \frac{1}{r} - r$. This is useful because when r is near to 1 then $\frac{1}{r} - r$ is less than $\frac{1}{2}$, satisfying the LHS of Proposition 8.

Proposition 9. $\Sigma_v |p(v)-q(v)| \leq \frac{1}{r} - r$.

We first prove

Lemma 7. $\frac{1}{r} - r = \frac{1}{r}\Sigma_v p(v) - \Sigma_v p(v, e == \texttt{ff})$.

Proof: $\frac{1}{r} - r = \frac{1}{r}\Sigma_v p(v) - r\Sigma_v q(v) = \frac{1}{r}\Sigma_v p(v) - r\Sigma_v \frac{p(v, e == \texttt{ff})}{r} = \frac{1}{r}\Sigma_v p(v) - \Sigma_v p(v, e == \texttt{ff})$. □

We then prove

Lemma 8. *for all v the following is true:*
$|p(v) - q(v)| \leq \frac{1}{r}p(v) - q_0(v)$
where $q_0(v) = p(v, \hat{e} == \texttt{ff})$.

This is proven by cases: (1) $p(v) \geq q(v)$: In this case $|p(v) - q(v)| = p(v) - q(v)$ and since $\frac{1}{r} \geq 1$ we have $p(v) - q(v) \leq \frac{1}{r}p(v) - q(v)$. Notice now that $q(v) \geq q_0(v)$ because $q(v) = P(\hat{C}_2 = v | \hat{e} == \texttt{ff}) \geq P(\hat{C}_2 = v, \hat{e} == \texttt{ff}) = q_0(v)$ (conditional probability is greater than joint probability): hence we conclude.

(2) $p(v) < q(v)$: In this case $|p(v) - q(v)| = q(v) - p(v)$. By definition $q(v) = \frac{1}{r}p(v, \hat{e} == \texttt{ff})$ and by basic probability $q_0(v) = p(v, \hat{e} == \texttt{ff}) \leq p(v)$ so we have $q(v) \leq \frac{1}{r}p(v)$ which gives us $q(v) - p(v) \leq \frac{1}{r}p(v) - p(v)$. Using again $p(v) \geq q_0(v)$ we conclude $q(v) - p(v) \leq \frac{1}{r}p(v) - p(v) \leq \frac{1}{r}p(v) - q_0(v)$ □

Combining the previous lemmas we now conclude the proof of the proposition:

for all v $|p(v) - q(v)| \leq \frac{1}{r}p(v) - q_0(v)$ implies
$\Sigma_v |p(v) - q(v)| \leq \Sigma_v (\frac{1}{r}p(v) - q_0(v)) = \frac{1}{r}\Sigma_v p(v) - \Sigma_v p(v, \hat{e} == \texttt{ff}) = \frac{1}{r} - r$.
□

Thus we can apply the $\mathcal{L}_1$ inequality and we get:

$$\begin{aligned} |\mathcal{H}(p) - \mathcal{H}(q)| &\leq \|p - q\|_1 (k - \log \|p - q\|_1) \\ &\leq (\frac{1}{r} - r)(k - \log(\frac{1}{r} - r)) \\ &\stackrel{\text{def}}{=} \mathcal{Z}(r) \end{aligned}$$

$\|p-q\|_1(k-\log\|p-q\|_1) \leq (\frac{1}{r}-r)(k-\log(\frac{1}{r}-r))$ is justified by Proposition 9 and the fact that $\lambda x.x(k - \log(x))$ is monotonic over the interval of interest, $(0, \frac{1}{2}]$, for $k \in \mathbb{N}^+$. This can be demonstrated by differentiating the function and determining the sign of the first derivative on the interval $(0, \frac{1}{2}]$ (always positive).

Hence, by noticing that $\mathcal{H}(p) = \mathcal{H}(\hat{C}_2)$ and $\mathcal{H}(q) = \mathcal{H}(\hat{C}_2 | \hat{e} == \texttt{ff})$ we get a lower bound on $\mathcal{H}(\hat{C}_2 | \hat{e} == \texttt{ff})$:

$$\mathcal{H}(\hat{C}_2 | \hat{e} == \texttt{ff}) \geq \mathcal{H}(\hat{C}_2) - \mathcal{Z}(r)$$

We can use this to obtain a lower bound for the command C as follows:

$$\begin{aligned} \mathcal{H}(\hat{C}) &\geq \mathcal{H}(\hat{C} | \hat{e}) \\ &= (1 - r)\mathcal{H}(\hat{C} | \hat{e} == \texttt{tt}) + r\mathcal{H}(\hat{C} | \hat{e} == \texttt{ff}) \\ &\geq r\mathcal{H}(\hat{C} | \hat{e} == \texttt{ff}) \\ &= r\mathcal{H}(\hat{C}_2 | \hat{e} == \texttt{ff}) \end{aligned}$$

$$\geq r(\mathcal{H}(\hat{C_2}) - \mathcal{Z}(r))$$
$$\geq s(a - \mathcal{Z}(s))$$

Where a is a lower bound for $\mathcal{H}(\hat{C_2})$ and $s \leq r$. To justify If(4) in Table 8 we are left with finding a lower bound $s \leq r$. In the rule this is given by solving $a_1 - b_2 = \mathcal{U}_k(1-s)$ where a_1 (resp. b_2) is a lower (resp. upper) bound for E_1(resp. E_2). This is a consequence of Proposition 7. In the rule, we restrict our search for the lower bound to the region $\frac{1}{2} \leq s \leq 1 - \frac{1}{2^k}$. This is safe since $s(a - \mathcal{Z}(s))$ is monotone in the region $1 - \frac{1}{2^k}$ to 1.

Notice that the quantities involved in inference rule [If(4)] can be automatically computed using techniques discussed in the analysis of boolean expressions.

Example The importance of the lower bounds provided by If(4) can be seen in the following example. Let P be the program

```
if  (h==n) {h=0} {y=h}
if  (y==m) {l=0} {l=1}
```

Assuming that the secret variable `h` is very unlikely to be equal to a constant `n` the true-branch of the first conditional will be chosen almost never and so almost always `y=h`. Hence statistically P is very similar to the program P'

```
if  (h==m ) {l=0} {l=1}
```

for an analysis to give similar results for P and P' we need the analysis of `y` in P at the end of the first conditional to be similar to the analysis of the secret input `h` which is `h`:[32,32] (we are assuming uniform distribution on the secret).

Using the same argument as in a previous example we have:
`[Eq2]` $\mathtt{h} : [32, 32] \vdash (\mathtt{h} == \mathtt{n}) : [0, \epsilon]$
where $\epsilon = \mathcal{B}(1/2^{32}) \approx 7.8 \times 10^{-9}$.

We also know that the analysis of the false-branch will be
`[Assign]` $\vdash \mathtt{h} : [32, 32]\{\mathtt{y} = \mathtt{h}\}\ \mathtt{y} : [32, 32]$
and so we get values $q = 1/2^{32}, a = 32$ which we use in the conditional rule to deduce
$\vdash \mathtt{h} : [32, 32]\{\mathtt{C}\}\mathtt{y} : [(1 - 1/2^{32})(32 - 4.2^{-8}), 32]$
where `C = if (h==n) {h=0} {y=h}`.
Notice that this bound is extremely close to [32,32] and hence the bounds for the low variable `l` in P and P' will be very close (in both cases around 7.8×10^{-9} bits).

On the other hand, suppose we didn't have good lower bounds for `y` after the first conditional (suppose for example we had $\mathtt{y} : [0, 32]$). Then testing `(y==m)` against a constant `m` will produce
`[Eq1]` $\mathtt{y} : [0, 32] \vdash (\mathtt{y} == \mathtt{m}) : [0, \mathtt{min}(1, 32)] = [0, 1]$.
This is because the side conditions for [Eq2] are not satisfied and so the only applicable rule is [Eq1] which will result in much higher bounds for `l` in P (1 bit) than in P' (around 7.8×10^{-9} bits).

5.3 Observing Non-termination

The so-called angelic semantics used in the correctness proof above has the effect of ignoring non-termination and hence any information flow which might flow as a result of variation in termination behaviour. Intuitively, this is dangerous. In reality, it is possible for a program's termination behaviour to vary according to its inputs, including the confidential ones. Moreover, given knowledge of the code, an observer can, in practice, observe that a program is never going to terminate. On the other hand, consider *how* an observer can tell that a program is stuck in an infinite loop: either by some low-level observation of the program state or by timing it. But our input-output model of observation - captured precisely by the simple state-transformer semantics - clearly excludes these possibilities. So there is a strong argument for ignoring non-termination in the proof: it falls outside our abstraction of the observer. Note, however, that this does *not* mean it is *safe* to ignore the possibility of such flows, only that modelling them would require a more fine-grained model of the observer (and hence a different semantics).

In fact, we can do a little better than this. Let us suppose that non-termination *is* observable. We can capture this by using the standard denotational semantics and treating $\perp$ as an observable value, like any other. How would this affect our results? The following result shows that, if we measure flows with respect to the standard denotational semantics, rather than the angelic semantics, our analysis under-estimates flows by at most one bit.

Proposition 10. *If* $\models \Gamma$ *and* $\vdash \Gamma\ \{C\}\ \Gamma'$ *then, for the transformational system induced by* $[\![C]\!]$, $\Gamma'(x) = [a,b] \Rightarrow \mathcal{F}_L(H \rightsquigarrow X) \leq (b+1)$, *where* $X = \lambda\sigma.\sigma(x)$.

Proof. Let $\mathcal{T}_C : \Sigma \to \{t, f\}$ be defined by:

$$\mathcal{T}_C\sigma = \begin{cases} t & \text{if } [\![C]\!]\sigma \neq \perp \\ f & \text{if } [\![C]\!]\sigma = \perp \end{cases}$$

Fix some program variable x and consider the corresponding output observations $\widehat{[\![C]\!]}$ and $\widehat{[\![C]\!]^A}$. By Shannon's inequalities, we have

$$\mathcal{H}(\widehat{[\![C]\!]}) \leq \mathcal{H}(\widehat{[\![C]\!]}, \mathcal{T}_C) = \mathcal{H}(\widehat{[\![C]\!]}|\mathcal{T}_C) + \mathcal{H}(\mathcal{T}_C) \tag{21}$$

Since (Lemma 2) $[\![C]\!]^A \sqsupseteq [\![C]\!]$, we have:

$$\widehat{[\![C]\!]}\sigma = \begin{cases} \widehat{[\![C]\!]^A}\sigma & \text{if } \mathcal{T}_C\sigma = t \\ \perp & \text{otherwise} \end{cases}$$

From this it follows that $\mathcal{H}(\widehat{[\![C]\!]}|\mathcal{T}_C = t) = \mathcal{H}(\widehat{[\![C]\!]^A}|\mathcal{T}_C = t)$, while $\mathcal{H}(\widehat{[\![C]\!]}|\mathcal{T}_C = f) = 0$. Hence, by the definition of conditional entropy, $\mathcal{H}(\widehat{[\![C]\!]}|\mathcal{T}_C) \leq \mathcal{H}(\widehat{[\![C]\!]^A}|\mathcal{T}_C)$. Thus $\mathcal{H}(\widehat{[\![C]\!]}|\mathcal{T}_C) \leq \mathcal{H}(\widehat{[\![C]\!]^A}|\mathcal{T}_C) \leq \mathcal{H}(\widehat{[\![C]\!]^A})$. Furthermore, since $\mathcal{T}_C$ is 2-valued, $\mathcal{H}(\mathcal{T}_C) \leq 1$. Hence, by (21), $\mathcal{H}(\widehat{[\![C]\!]}) \leq \mathcal{H}(\widehat{[\![C]\!]^A}) + 1$. The proposition then follows by Corollary 2. □

6 Analysis of arithmetic expressions

We can improve the analysis of leakage via arithmetic expressions by exploiting algebraic knowledge of the operations together with information about the operands acquired through supplementary analyses such as parity analysis, constant propagation analysis etc. The (binary) operations we consider are addition, subtraction, multiplication $(+, -, *)$ on the twos-complement representations of k bit integers with overflow.

We use $\odot$ for the binary operator while the random variables X, Y and Z range over the first and second inputs and the output of the operator respectively. They are related by

$$P(Z = z) = \sum_{(x,y)\in\odot^{-1}(z)} P(X = x, Y = y)$$

We assume we know bounds on the entropy of the input space, $\mathcal{H}(X, Y)$, and entropy of the projected input spaces, $\mathcal{H}(X)$ and $\mathcal{H}(Y)$ and we aim to find bounds on the entropy of the output space, $\mathcal{H}(Z)$.

Since a binary arithmetic operation on twos-complement integers is a function from $X \times Y$ to Z and since functions can only reduce entropy or leave it the same we have

$$0 \leq \mathcal{H}(Z) \leq \mathcal{H}(X, Y)$$

In general we will not know $\mathcal{H}(X, Y)$ but only $\mathcal{H}(X)$ and $\mathcal{H}(Y)$. Since $\mathcal{H}(X, Y) \leq \mathcal{H}(X) + \mathcal{H}(Y)$ we can use this sum as an upper bound. The upper bound observation is captured for an operation $\odot$ in the rule [OpMax] in table 11.

$$[\text{OpMax}]\ \frac{\Gamma \vdash E_1 : [-, b_1]\ \ \Gamma \vdash E_2 : [-, b_2]}{\Gamma \vdash E_1 \odot E_2 : [0, b_1 + b_2]}$$

$$[\text{AddMin}]\ \frac{\Gamma \vdash E_1 : [a_1, b_1]\ \ \Gamma \vdash E_2 : [a_2, b_2]}{\Gamma \vdash E_1 + E_2 : [\max(a_1, a_2)\ -\ \min(b_1, b_2), k]}$$

$$[\text{ConstAdd}]\ \frac{E_1 \text{ is constant}\ \ \Gamma \vdash E_2 : [a, b]}{\Gamma \vdash E_1 + E_2 : [a, b]}$$

$$[\text{ZeroMult}]\ \frac{E_1 \text{ is zero}\ \ \Gamma \vdash E_2 \text{: } [a, b]}{\Gamma \vdash E_1 * E_2 : [0, 0]}$$

$$[\text{OddMult}]\ \frac{E_1 \text{ is an odd constant}\ \ \Gamma \vdash E_2 : [a, b]}{\Gamma \vdash E_1 * E_2 : [a, b]}$$

Table 11. Some Refined Analysis rules

Further improvements to either the upper or the lower bound depend on knowledge more specific to the operation and/or the expressions.

Before examining individual operations there is something we can say about the relationship between $\mathcal{H}(Z)$ and $\mathcal{H}(X,Y)$ that holds for all operations which we exploit directly in Proposition 13.

We can use the functional relationship between inputs and outputs to show that the entropy of the output space is the entropy of the input space less the entropy of the input space *given knowledge of the output space*. This latter quantity is a measure of the entropy of the input space destroyed by the function $\odot$. The idea is expressed formally in the following proposition:

Proposition 11. *Let $Z = \odot(X,Y)$ then $\mathcal{H}(Z) = \mathcal{H}(X,Y) - \mathcal{H}(X,Y|Z)$.*

Proof. Using conditional entropy expressions we have this relationship between $\mathcal{H}(X,Y)$ and $\mathcal{H}(Z)$: $\mathcal{H}(X,Y,Z) = \mathcal{H}(Z) + \mathcal{H}(X,Y|Z)$. However we also have

$$\begin{aligned} p(x,y,z) &= p(x,y) \text{ if } (x,y) \in \odot^{-1}(z) \\ &= 0 \text{ otherwise} \end{aligned}$$

Hence

$$\begin{aligned} \mathcal{H}(X,Y,Z) &= -\sum_{x,y,z} p(x,y,z)\log(p(x,y,z)) \\ &= -\sum_{x,y} p(x,y)\log(p(x,y)) \\ &= \mathcal{H}(X,Y) \end{aligned}$$

Then we have $\mathcal{H}(X,Y) = \mathcal{H}(Z) + \mathcal{H}(X,Y|Z)$ and so $\mathcal{H}(Z) = \mathcal{H}(X,Y) - \mathcal{H}(X,Y|Z)$. □

6.1 Addition and subtraction

Addition and subtraction are essentially the same operation in twos-complement arithmetic so we restrict our attention to addition.

Bitwise addition $(+)$ makes the set of numbers representable in twos-complement using k bits a cyclic additive group with identity 0 and generator 1 (so $a+1$ has its usual meaning except when $a = 2^{k-1}-1$, in which case $a+1 = -2^{k-1}$). The inverse $-a$ is given by the twos-complement operation (so $-a$ has its usual meaning except for $a = -2^{k-1}$, which is its own inverse).

Proposition 12. *Let $T_k = \{-2^{k-1}, \ldots, 2^{k-1}-1\}$, the set of integers representable by k bits in twos-complement. Bitwise addition $(+)$ makes T_k a cyclic additive group with identity 0 and generator 1.*

We don't include a proof here but the proposition is straightforward to verify.

As an immediate consequence, addition of a constant is just a permutation on T_k and thus leaves entropy unchanged. This is captured by rule [ConstAdd]. The symmetric rule which follows from commutativity of addition is left implicit.

The cyclic group structure further allows us to show that either operand of + is a function of the other operand and the result. This in turn allows us to establish a tighter lower bound for +: the entropy of the outcome, $\mathcal{H}(Z)$, is bigger than or equal to the entropy of the input space, $\mathcal{H}(X,Y)$ less the smaller of the two projected entropies for that space, $\mathcal{H}(X)$ and $\mathcal{H}(Y)$.

Proposition 13. *Let* $Z = +(X,Y)$*, then*

$$\mathcal{H}(Z) \geq \mathcal{H}(X,Y) - min(\mathcal{H}(X), \mathcal{H}(Y))$$

Proof. First we establish that $\mathcal{H}(Z) \geq \mathcal{H}(X,Y) - \mathcal{H}(X)$.

We can make arguments similar to the one employed to justify Proposition 11 and demonstrate that $\mathcal{H}(X,Y,Z) = \mathcal{H}(X,Z)$.

By Proposition 11 we have $\mathcal{H}(Z) = \mathcal{H}(X,Y) - \mathcal{H}(X,Y|Z)$ so it suffices to show that $\mathcal{H}(X,Y|Z) \leq \mathcal{H}(X)$

$$\begin{aligned} & \mathcal{H}(X,Y|Z) \\ &= \mathcal{H}(X,Y,Z) - \mathcal{H}(Z) \\ &= \mathcal{H}(X,Z) - \mathcal{H}(Z) \\ &= \mathcal{H}(X|Z) \\ &\leq \mathcal{H}(X) \end{aligned}$$

By a similar argument we can also establish $\mathcal{H}(Z) \geq \mathcal{H}(X,Y) - \mathcal{H}(Y)$. These two inequalities establish the proposition. □

Since $\mathcal{H}(X,Y) \geq \max(\mathcal{H}(X), \mathcal{H}(Y))$ we can safely replace $\mathcal{H}(X,Y)$ with that quantity. This provides the rule [AddMin] in table 11 for calculating an improved lower bound for addition.

6.2 Multiplication

Multiplication is less straightforward to analyse than addition as the algebraic structure of the operation is more complex. We are not currently able to provide any general result for the operation. However, in the event that some subsidiary prior analysis is able to identify a useful property of one of the operands of the operation, we can get very good bounds on the entropy of the output, in particular when one of the operands is odd or when one of the operands is zero.

Multiplication by zero always has zero as the result, i.e. Z has value space singleton set $\{0\}$ whose element has probability 1, so knowing that one operand is zero guarantees that $\mathcal{H}(Z) = 0$. This observation is captured in the rule [ZeroMult] in table 11. The symmetric rule which follows from commutativity of multiplication is left implicit.

To justify the rule [OddMult] recall that the order of a group is the number of its elements, hence the order of T_k is 2^k. Furthermore, the order of any element a is defined to be the order of the cyclic subgroup $<a>$ which it generates (with elements $\{n.a : n \geq 0\}$). We denote the order of an element a by $o(a)$. We can then state the following

Proposition 14. *For $a \in T_k$ where a is odd, $o(a) = 2^k$, i.e. any odd element is a generator for T_k.*

This can be demonstrated using some elementary group theory since every odd number does not divide the order of T_k.

Because odd elements are generators for the whole set, multiplication by an odd constant (i.e. zero entropy in one component) can be viewed as an injective function from T_k to T_k and so the entropy of the output space can never be less than the entropy of the other operand.

Proposition 15. *Let $Z = *(X, Y)$ where $\forall x \in X.\ x$ is an odd constant, then*

$$\mathcal{H}(Z) \geq \mathcal{H}(Y)$$

.

The implication of this result for propagation of lower bounds during multiplication is captured in the rule [OddMult] in table 11. The symmetric rule which follows from commutativity of multiplication is left implicit.

Although we don't have anything we can say in general about leakage via multiplication we do have a theoretical result which may prove useful in future work: we know the size of the inverse image of an element of T_k.

Theorem 3. *For c in T_k,*

$$|*^{-1}(c)| = \begin{cases} (k + 1 - \log o(c))2^{k-1} & \text{if } c \neq 0 \\ (k+2)2^{k-1} & \text{if } c = 0 \end{cases}$$

7 Conclusions and future work

The work presented in this paper is the first time information theory has been used to measure interference between variables in a simple imperative language with loops. An obvious and very desirable extension of the work would be to a language with probabilistic operators.

Incremental improvement of the analysis could be given by a subtler treatment of loops and by improved bounds on a wider range of expressions. A similar syntax based, security related analysis might be applied to queries on a secure database. Denning [8] did work on information flow in database queries.

It would be also interesting to be able to provide a "backward" analysis where given an interference property that we want a program to satisfy we are able to deduce constraints on the interference of the input. A simple example of this scenario is provided by Shannon's perfect secrecy theorem where the property of non-interference on the output implies the inequality $\mathcal{H}(L) \geq \mathcal{H}(\texttt{Out})$.

Timing issues like "rate of interference" could also be analysed by our theory allowing for a quantitative analysis of "timing attacks" [31].

On a more speculative level we hope that quantified interference could play a role in fields where modularity is an issue (for example model checking or

information hiding). At the present modular reasoning seems to break down whenever modules interfere by means other than their interfaces. However if once quantified the interference is shown to be below the threshold that affects the desired behaviour of the modules, it could be possible to still use modular reasoning. An interesting development in this respect could be to investigate the integration of quantified interference with non-interference based logic [23, 11].

In our work information theory and denotational semantics provide two different levels in the analysis. However recent work relating denotational semantics and entropy has shown that there is an interaction between these two disciplines: both theories model in different albeit related ways partiality/uncertainty. The choice of flat domains as a denotational semantics for our programming language was motivated by a desire to emphasise that our analysis assumes programs are not reactive but have a functional, or data transforming, semantics. It may well be an accident that these flat domains sit comfortably with entropic measures but the work of K. Martin [15] begs the question whether there is a more fundamental relationship between the two being utilised in our work.

The authors would like to thank Peter O'Hearn and the anonymous referees for their helpful comments, corrections and suggestions for improvement.

References

1. Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In Roberto Giacobazzi, editor, *SAS 2004 (11th Static Analysis Symposium), Verona, Italy, August 2004*, volume 3148 of *LNCS*, pages 100–115. Springer-Verlag, 2004.
2. K.J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronics Systems Division, Bedford, MA, April 1977.
3. D. Clark, S. Hunt, and P. Malacaria. Quantified interference for a while language. In *Electronic Notes in Theoretical Computer Science 112*, pages 149 – 166. Elsevier, 2005.
4. David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative analysis of the leakage of confidential data. In Alessandra Di Pierro and Herbert Wiklicky, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier, 2002.
5. Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Belief in information flow. In *18th IEEE Computer Security Foundations Workshop*, 2005.
6. Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley Interscience, 1991.
7. D. E. R. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5), May 1976.
8. D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
9. J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.
10. Sebastian Hunt and David Sands. On flow-sensitive security types. In *Proc. Principles of Programming Languages, 33rd Annual ACM SIGPLAN - SIGACT Symposium (POPL'06)*, Charleston, South Carolina, USA, January 2006. ACM Press. To appear.
11. S. Isthiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *28th POPL*, pages 14–26, London, January 2001.

12. Richard L.Burden and J. Douglas Faires. *Numerical Analysis*. PWS-KENT, 1989. ISBN 0-534-93219-3.
13. Gavin Lowe. Quantifying information flow. In *Proceedings of the Workshop on Automated Verification of Critical Systems*, 2001.
14. Pasquale Malacaria. Assessing security threats of looping constructs. In *Principles of Programming Languages*, 2007.
15. Keye Martin. Entropy as a fixed point. *Theoretical Compututer Science*, 350(2–3):292–324, 2006.
16. James L. Massey. Guessing and entropy. In *Proc. IEEE International Symposium on Information Theory*, Trondheim, Norway, 1994.
17. Annabelle McIver and Carroll Morgan. A probabilistic approach to information hiding. In *Programming methodology*, pages 441–460. Springer-Verlag New York, Inc., New York, NY, USA, 2003.
18. John McLean. Security models and information flow. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, Oakland, California, May 1990.
19. Jonathan Millen. Covert channel capacity. In *Proc. 1987 IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society Press, 1987.
20. C.C. Morgan, A.K. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming and Systems*, 18(3):325–353, May 1996.
21. Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Approximate non-interference. In Iliano Cervesato, editor, *CSFW'02 – 15th IEEE Computer Security Foundation Workshop*. IEEE Computer Society Press, June 2002.
22. Gordon Plotkin. Lambda definability and logical relations. Technical Report Memorandum SAI-RM-4, Department of Artificial Intelligence, University of Edinburgh, 1973.
23. J. Reynolds. Separation logic: a logic for shared mutable data structures. Invited Paper, LICS'02, 2002.
24. J. C. Reynolds. Syntactic control of interference. In *Conf. Record 5th ACM Symp. on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, 1978. ACM, New York.
25. Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. In *Proc. European Symposium on Programming*, Amsterdam, The Netherlands, March 1999. ACM Press.
26. Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. 13th IEEE Computer Security Foundations Workshop*, Cambridge, England, July 2000. IEEE Computer Society Press.
27. Claude Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948. Available on-line at http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html.
28. Richard Statman. Logical relations and the typed lambda calculus. *Information and Control*, 65, 1985.
29. D. Sutherland. A model of information. In *Proceedings of the 9th National Computer Security Conference*, September 1986.
30. D. Volpano and C. Irvine. Secure flow typing. *Computers and Security*, 16(2):137–144, 1997.
31. Dennis Volpano. Safety versus secrecy. In *Proc. 6th Int'l Symposium on Static Analysis*, LNCS, pages 303–311, Sep 1999.
32. Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pages 156–168, Rockport, MA, 1997.

33. Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *Proc. 27th ACM Symposium on Principles of Programming Languages*, pages 268–276, Boston MA, Jan 2000.
34. Raymond W. Yeung. A new outlook on shannon's information measures. *IEEE Transactions on Information Theory*, 37(3), May 1991.